MASTER EN SCIENCES INFORMATIQUES

Automatisation du développement d'applications hautement interactives dans un langage de quatrième génération en utilisant le concept de framework

Darville, Christophe

Award date:
1994

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

Automatisation du développement d'applications hautement interactives dans un langage de quatrième génération en utilisant le concept de framework

Christophe DARVILLE

Promoteur : Professeur F. Bodart

Mémoire présenté en vue de l'obtention du titre de Licencié et Maître en Informatique

Année académique 1993-1994

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21B
B-5000 Namur

Automatisation du développement d'applications hautement interactives dans un langage de quatrième génération en utilisant le concept de framework

Christophe DARVILLE

Résumé

L'objectif de ce mémoire est de présenter un outil permettant d'automatiser le développement d'applications ayant une interface graphique. Les applications produites sont ensuite récupérées dans un environnement de développement afin de pouvoir être améliorées. Le développement d'une application consiste à personnaliser un framework qui correspond à une application générique. Une partie de cette personnalisation sera alors réalisée automatiquement grâce à des informations conceptuelles fournies à travers l'utilisation d'un outil CASE.

Abstract

The subject of this thesis is the description of a tool which automate the engineering of softwares with graphical user interfaces. Once the software is produced, it can be improved with development tools. The engineering of an application consists in the customization of a framework. A framework can be seen as a generic application. A part of the customization will be made automatically on the basis of conceptual information obtained from a tool CASE.

Mémoire présenté en vue de l'obtention du titre de
Licencié et Maître en Informatique

Septembre 1994

Promoteur : Professeur F. Bodart

Je tiens à remercier mon promoteur, Monsieur le Professeur Bodart, en qui j'ai trouvé un soutien tout au long de ce mémoire.

Je souhaite également exprimer ma reconnaissance à Monsieur André Gonay, mon maître de stage, qui m'a apporté son aide pour la réalisation de ce mémoire chaque fois que j'en avais besoin.

Je désire aussi remercier toute l'équipe d'Intègre où j'ai effectué mon stage, et plus particulièrement France et David.

Derrière ce travail se trouvent aussi de nombreuses personnes qui, de près ou de loin, m'ont aidé et encouragé; je pense en particulier à Natalia et Christelle. Que ces personnes trouvent ici tous mes remerciements.

A mes parents, qui m'ont permis de réaliser ces études.

Table des matières

Introduction

Introduction

A l'heure actuelle, l'utilisateur est de plus en plus pris en considération lors du développement d'une application. Tant en ce qui concerne la recherche d'une meilleure interface homme-machine pour l'application (facilité d'utilisation et convivialité) qu'une meilleure prise en compte de ses besoins.

Pour rendre les applications plus faciles et plus agréables à utiliser, l'accent a été mis sur les qualités de l'interface utilisateur. En effet, ce sont elles qui déterminent la facilité d'apprentissage, la facilité et l'efficacité d'utilisation. Elles se traduisent directement en terme d'acceptation, de satisfaction, de productivité pour l'utilisateur, de rentabilité pour l'entreprise ([Mein91]).

Aujourd'hui, bon nombre d'applications présentent une interface graphique. Il s'agit d'interfaces qui permettent à l'utilisateur de manipuler directement des objets sur l'écran, et cela, principalement par l'intermédiaire d'une souris.

Grâce à cette manipulation directe, l'utilisateur crée, modifie, manipule des objets à l'écran avec l'impression d'être sans autre intermédiaire que ce qu'il voit à l'écran ([Mein91]).

Ces objets peuvent être des boîtes de dialogue, des boutons de commande, des boîtes à cocher, des boutons-radio, des listes de sélection, des champs d'édition uni-linéaire, des champs d'édition multi-linéaire, ... On trouvera une description détaillée des différents objets interactifs dans [Sacr92].

De plus, avec ce genre d'interface, l'utilisateur travaille sur des icônes (représentation idéographiques, sur l'écran, des objets dans l'ordinateur) qui, si elles sont bien choisies permettent de déduire naïvement les fonctions et attributs des objets qu'elles représentent, ainsi que les opérations que l'on peut leur faire subir ([Mein91]).

On veut en fait travailler avec des objets qui soient les plus compréhensibles intuitivement par l'utilisateur.

En ce qui concerne une meilleure prise en compte des besoins de l'utilisateur, on travaille maintenant de plus en plus avec des prototypes que l'on présente à l'utilisateur et qui lui permettent d'ajuster ses besoins en cours de développement de l'application.

On agit de la même façon qu'un architecte qui présente à ses clients une maquette en trois dimensions de leur future maison plutôt qu'un plan sur papier. Il devient plus facile pour eux de se représenter ce que sera la maison et donc plus facile de voir ce qu'ils aimeraient changer avant que la maison ne soit construite et qu'il ne soit trop tard.

La création de prototypes pour les applications graphiques est rendue plus facilement réalisable grâce aux nouveaux outils de développement d'applications graphiques que l'on appelle aussi langages de quatrième génération (L4G). Ces outils permettent de développer rapidement et efficacement des applications graphiques qui sont généralement liées à une base de données. On peut citer des outils tels que PowerBuilder, NS-DK, SQLWindows, Visual Basic, ...

On le voit, l'accent est mis sur le développement d'applications qui satisfont les utilisateurs.

Mais de plus, on veut également que ces applications soient développées le plus rapidement possible. Il est clair qu'à travers cela, on poursuit principalement des objectifs économiques d'augmentation de productivité des utilisateurs et de réduction des coûts de production des applications.

C'est pour répondre à ce double objectif de satisfaction des utilisateurs et de rapidité de développement des applications que nous allons envisager dans ce mémoire l'automatisation du développement d'applications interactives ayant une interface graphique dans un langage de quatrième génération en utilisant le concept de framework.

A travers ce concept de framework, on veut en fait mettre en oeuvre, comme nous allons le voir, un processus de réutilisation de composants. Or, on croit fortement que la réutilisation est une solution permettant d'augmenter la productivité et la qualité lors du développement de logiciels ([Bigg87]).

En fait, un framework représente une solution générique pour une classe de problèmes donnés, généralement appartenant au même domaine, et qui peut être personnalisée pour répondre à un problème particulier ([Beck93]).

Au vu de cette définition, on constate qu'il y a deux grands rôles distincts dans le développement d'une application. Il y a, d'une part, l'ingénieur d'application (*application engineer*) qui est la personne chargée de réaliser le framework et,

d'autre part, le développeur d'application (*application developer*) qui est la personne qui va personnaliser le framework ([Beck93]).

La personnalisation d'un framework avec le rôle du développeur d'application joué par un utilisateur final ou par une personne proche de l'utilisateur, devrait permettre d'obtenir *rapidement* une application qui soit telle que ce dernier la désire, c'est à dire qui *réponde réellement à ses besoins*.

Dans la suite de ce travail, nous allons discuter des nouveaux outils de développement d'applications graphiques que l'on qualifie de L4G. Nous allons voir en quoi ils constituent des L4G, en donnant tout d'abord une définition des L4G. Après quoi nous présenterons le concept de framework. Nous détaillerons finalement l'outil PowerTalk qui permet d'automatiser le développement d'applications fonctionnant sous MS-Windows dans le L4G PowerBuilder en utilisant le concept de framework.

Chapitre 1

Chapitre 1 : Les langages de quatrième génération

1.1. Introduction

Depuis quelques années, on a vu apparaître sur le marché une nouvelle catégorie de langages permettant de développer rapidement des applications de gestion interactives ayant une interface graphique.

Dans la suite de ce mémoire, nous nous limiterons à considérer les applications de gestion, et lorsque nous utiliserons le terme application, il s'agira d'applications de gestion. Il s'agit d'applications dans lesquelles les fonctions manipulent une base de données ([Sacr91]).

En fait, il s'agit d'outils permettant de créer à la fois une application avec interface graphique ainsi que la structure de la base de données sur laquelle elle va travailler. Avec ce genre d'outil, l'accès aux données à partir d'une application est souvent rendu très facile à programmer. Les principaux outils de ce style sont PowerBuilder, NS-DK, Visual Basic, ObjectView et SQLWindows.

Cependant, bien qu'il s'agisse de véritables environnements de développement, nous allons voir en quoi ils constituent réellement des L4G.

Dans le chapitre suivant, nous allons faire une description de ces nouveaux outils de développement après quoi nous montrerons que ces outils constituent bien des L4G.

Mais avant cela, nous allons consacrer ce chapitre aux L4G en général. Nous allons tout d'abord montrer ce que les L4G apportent par rapport aux langages de troisième génération. Après quoi nous donnerons une définition des L4G. Nous terminerons par une description des différentes catégories de L4G.

1.2. Apports des langages de quatrième génération

Pendant presque vingt-cinq ans, les langages de programmation utilisés pour développer des applications de gestion ont évolués très lentement. Il y avait des différences de syntaxes, mais les constructions de base de Cobol, Pascal, PL/I, ... étaient similaires (branchement, boucles, ...). Il s'agissait des langages de troisième génération.

Depuis le début des années 80 est apparue une nouvelle génération de langages. Ce sont les langages de quatrième génération. A la différence des langages de troisième génération, la variété de style au sein des L4G est très étendue.

En effet, comme nous le verrons, il y a des L4G qui s'adressent aux utilisateurs finaux tandis que d'autres s'adressent plutôt aux programmeurs professionnels. Il y en a qui proposent beaucoup de fonctionnalités tandis que d'autres ne fournissent qu'un nombre limité de fonctionnalités. Il y en a qui sont exclusivement non-procéduraux tandis que d'autres n'ont qu'une partie non-procédurale.

Les langages de troisième génération tels que Cobol, Fortran, Pascal, C, ... sont des langages indépendants du matériel jusqu'à un certain point.

On les appelle langages de *haut-niveau* car une instruction correspond en général à plusieurs instructions en langage machine, tandis qu'une instruction en assembleur (langage de deuxième génération) correspondait en général à une instruction en langage machine ([Keye92], [Mart85]). De plus, la syntaxe des instructions est en général assez proche de l'anglais et les formules sont exprimées de façon mathématique.

Il est bien évidemment plus facile d'écrire :

X = (A + B) / (C + D)

que d'écrire dans un langage assembleur quelque chose comme :

```
CLA C
ADD D
STO Y
CLA A
ADD B
DIV Y
STO X
```

Cependant, les programmes conçus à l'aide de langages de troisième génération demandent encore beaucoup de lignes de code et ne peuvent être

réalisés que par des programmeurs professionnels. De plus, ils sont longs à déboguer et la modification de systèmes complexes est très difficile ([Mart85]).

Pour palier les limitations des langages de troisième génération, on est passé à une nouvelle génération de langages qui sont les langages de quatrième génération.

On qualifie ces langages de langages de *haute productivité,* et ils ont été conçus selon [Mart85] et [Keye92] pour :

- accélérer le processus de développement d'applications
- rendre les applications plus facilement et plus rapidement modifiables, réduisant ainsi les coûts de maintenance
- fournir des macro-instructions qui réduisent le nombre d'instructions que doit fournir un programmeur
- être plus proche des utilisateurs de sorte qu'ils puissent résoudre eux-mêmes leurs propres problèmes
- accélérer le processus de débogage.

Nous allons maintenant donner une définition des langages de quatrième génération.

1.3. Définition des langages de quatrième génération

A vrai dire il n'est pas évident de fournir une définition claire et précise de ce qu'est un L4G. Comme nous allons le voir, il en existe une grande diversité qui ont des champs d'application parfois très différents.

Selon [Buck90], un L4G peut être vu comme étant n'importe quel outil de programmation ou de développement de systèmes qui permet le **développement rapide d'une application**.

Certains auteurs prétendent qu'un L4G peut augmenter jusqu'à dix fois la vitesse de développement d'une application ([Buck90], [Keye92]).

On constate que par L4G, on n'entend pas uniquement les langages au sens strict du terme, mais on entend aussi les outils permettant de développer une application.

En fait, plutôt que de tenter de donner une définition d'un L4G nous allons relever les caractéristiques principales des L4G. Il est cependant bon de noter que tous les L4G n'auront pas toutes ces caractéristiques, et que la liste n'est pas exhaustive ([Buck90], [Keye92]).

1.3.1. Langage non-procédural

On qualifie souvent les L4G de langages non-procéduraux ([Buck90], [Keye92], [Mart85]).

Un langage procédural spécifie **comment** accomplir quelque chose, tandis qu'un langage non-procédural spécifie **quoi** accomplir, sans décrire comment ([Mart85]).

Un langage tel que Cobol est procédural. En effet, les programmeurs doivent donner précisément les instructions détaillées spécifiant comment est accomplie chaque action. Tandis que dans un langage non-procédural, l'utilisateur dit principalement ce qu'il faut faire sans se préoccuper de la façon dont ce sera réalisé.

Employé	Numéro	Nom	Salaire	Département

Figure 1.1 : table employé

On peut prendre comme exemple de langage non-procédural le langage QBE (Query-By-Example) d'IBM. Il s'agit d'un langage de requête très facile à utiliser.

En fait, l'utilisateur va travailler directement à l'écran sur des représentations de tables. Mais nous allons prendre un exemple pour illustrer notre propos.

Si l'on considère que l'utilisateur dispose à l'écran de la table représentée à la figure 1.1.

Il s'agit d'une table correspondant à une liste d'employés. Chaque employé a un numéro, un nom, un salaire et il travaille dans un certain département.

Supposons que l'utilisateur désire obtenir le nom de tous les employés qui travaillent dans le département commercial. Alors, il va remplir la table employé comme indiqué dans la figure 1.2.

Employé	Numéro	Nom	Salaire	Département
		P.		Commercial

Figure 1.2 : table employé contenant la requête de l'utilisateur

Le code P. correspond à un opérateur de QBE qui signifie qu'il faut afficher les données de la colonne Nom qui satisfont à la requête. QBE dispose d'autres opérateurs permettant notamment de mettre à jour, d'insérer ou encore de supprimer des données. On trouvera dans [Mart86] une description détaillée du langage QBE et de ses différents opérateurs.

Lorsque l'utilisateur aura fourni sa requête, il en demandera l'exécution. Il obtiendra alors le résultat de la requête comme il est représenté à la figure 1.3.

Employé	Numéro	Nom	Salaire	Département
		Dupont **Durand** **Martin**		**Commercial**

Figure 1.3 : table employé contenant le résultat de la requête

On voit donc clairement qu'avec un langage non-procédural, l'utilisateur demande ce qu'il veut obtenir sans se soucier de comment l'obtenir. Ici, l'utilisateur veut obtenir la liste des employés qui travaillent dans le département commercial, et il ne se préoccupe pas de savoir comment QBE va se charger de l'obtenir.

La plupart des meilleurs L4G combinent à la fois les avantages de la programmation non-procédurale et ceux de la programmation procédurale. Même dans des systèmes complexes, une bonne partie du travail peut être fait de façon non-procédurale, comme par exemple la génération d'écrans, la génération de rapports ou encore l'accès aux données ([Mart85], [Buck90]).

1.3.2. Fonctionnalités limitées

Les L4G varient beaucoup dans leur puissance et dans leur capacité. Il y en a qui sont de simples langages de requête, d'autres qui permettent uniquement de générer des rapports ou des écrans. Cependant, il y en a d'autres qui sont des langages de programmation de haut-niveau qui permettent de générer des applications complètes.

Alors que les langages de troisième génération permettaient de créer quasiment toutes les applications, et même de la plus haute complexité, il faut remarquer que certains L4G sont conçus uniquement pour des classes d'applications spécifiques qui peuvent parfois être très restreintes.

On constate donc qu'avec les L4G beaucoup plus qu'avec les langages de troisième génération, il est nécessaire de choisir le langage qui correspond le mieux à l'application que l'on veut réaliser ([Mart85]).

1.3.3. Simplicité d'utilisation

En fait, cette caractéristique est liée aux deux précédentes. Puisque certains L4G ne proposent d'une part, comme on vient de le voir, qu'un nombre limité de fonctionnalités, il est clair qu'ils sont plus rapidement étudiés et plus facilement utilisables. D'autre part, puisqu'ils sont largement non-procéduraux, leur utilisation est plus simple et plus intuitive.

De plus, certains L4G sont conçus de façon à ce qu'il y ait un véritable dialogue entre l'utilisateur et le système lorsque l'utilisateur est en train de développer son application. Ce n'était pas le cas avec les langages de troisième génération où le programmeur était seul et n'avait aucune guidance lorsqu'il écrivait son programme.

Ce dialogue peut se faire sous la forme d'un simple échange de messages. Mais on peut aussi avoir, et c'est de plus en plus souvent le cas, une interaction utilisant la pleine potentialité du graphisme, du fenêtrage, de la manipulation de la souris, ... L'utilisateur pourra par exemple interagir à l'aide de menus, remplir des panneaux affichés à l'écran, manipuler les données directement dans une fenêtre, ...

On qualifie souvent les L4G de langages proches de l'utilisateur (*user-friendly*) parce que certains L4G, de par leur simplicité d'utilisation, sont conçus pour être utilisés autant par des utilisateurs finaux que par des programmeurs professionnels.

1.3.4. Facilité de débogage

Le débogage est effectué au niveau du L4G et jamais au niveau du code objet.

1.3.5. Facilité de maintenance

Les applications peuvent être rapidement changées au niveau du L4G ce qui facilite la maintenance. Il n'y a aucun travail de maintenance qui est réalisé au niveau du code objet.

Mais ce qui est plus important, c'est que les L4G sont en général des outils qui sont appropriés au prototypage. Nous verrons en détail au point 2.2.4. ce qu'on entend exactement par prototypage et les avantages que cela procure. Cependant, on peut déjà souligner qu'en utilisant correctement cette approche, on peut obtenir une application qui satisfait mieux aux besoins de l'utilisateur.

Dès lors, si l'application satisfait mieux aux besoins de l'utilisateur, les efforts de maintenance qui consistent souvent à adapter l'application aux demandes de l'utilisateur pourront être réduits fortement.

1.3.6. Facilité de la manipulation des données

La plupart des L4G offrent des interfaces simples avec les systèmes de gestion de bases de données. Le programmeur a en général peu de préoccupation en ce qui concerne la manipulation des données.

Concrètement, les données vont souvent apparaître à l'utilisateur sous une forme logique. Si on reprend le langage QBE, on constate que les données apparaissent à l'utilisateur sous la forme logique de tables. Toutes les manipulations des données qu'il va effectuer se feront en utilisant les tables. Il pourra de la sorte consulter, insérer, supprimer ou encore modifier les données.

L'utilisateur n'aura aucun souci de savoir comment les données sont stockées physiquement, et sur la manière d'y accéder.

C'est le L4G qui se chargera de faire les transformations entre la représentation logique des données et leur stockage physique.

Dans le chapitre suivant, nous verrons un exemple d'interface avec les données dans le L4G PowerBuilder. Cette interface étant réalisée grâce au concept de datawindow.

1.3.7. Intégration des aspects interface, traitements et données

En général, les L4G fournissent un environnement où les aspects interface, traitements et données sont intégrés. Il ne faut pas sortir de l'environnement de travail pour, par exemple, accéder aux données.

Cette intégration n'est pas le cas avec les langages de troisième génération. Par exemple, lorsque l'on veut accéder en Cobol à des données stockées sur une base de données relationnelles, il est nécessaire d'intégrer soi-même une requête SQL dans le corps du programme, et il faudra prendre en charge la récupération de ces données dans des variables qu'il aura fallu définir au préalable. Cela causant parfois de grandes difficultés au niveau de la compatibilité des types de données que l'on ramène dans les variables de Cobol. Comme on l'a vu au point précédent, tout est pris en charge par le L4G.

De même, les différents traitements que l'on peut effectuer sur les données sont intégrés dans l'environnement.

On a vu un exemple d'une telle intégration avec le langage QBE.

1.4. Catégories de L4G

Nous allons maintenant rapidement passer en revue les principales catégories de L4G existantes ([Buck90], [Keye92] et [Mart85]).

1.4.1. Les langages de requêtes

On a vu un exemple d'un tel langage avec QBE, on peut également citer comme autre exemple le langage SQL, ou encore l'outil Quest de Gupta fonctionnant sous MS-Windows et permettant d'accéder aux systèmes de gestion de bases de données relationnelles les plus utilisés.

Avec un outil comme Quest, un utilisateur peut, en principe, fournir sa requête en utilisant la souris sans devoir connaître le langage SQL.

Les langages de requêtes sont généralement faciles à utiliser malgré que parfois, la requête puisse entraîner pour le L4G une recherche complexe dans la base de données qui impliquera plusieurs enregistrements ou tables.

La plupart des langages de requêtes permettent, outre le fait de pouvoir consulter les données, d'insérer de nouvelles données, de supprimer certaines données ou encore de les modifier.

1.4.2. Les générateurs de rapports

Il s'agit d'outils permettant d'extraire rapidement des données d'une base de données ou de fichiers afin de les insérer dans un rapport.

Un bon générateur de rapport sera capable de joindre ensemble plusieurs enregistrements lorsqu'ils ne constituent qu'une et une seule donnée demandée par l'utilisateur et d'insérer cette donnée "agrégée" dans le rapport.

1.4.3. Les langages d'aide à la décision

Ces outils permettent d'assister le décideur dans sa prise de décision. Ils fournissent en général la possibilité d'inclure rapidement des analyses statistiques ou des analyses de tendance, ou encore des analyses de type *"what-if"*. On peut citer comme exemple de tels outils les tableurs actuels (par exemple Excel 5).

1.4.4. Les générateurs d'application

Il s'agit de la catégorie des L4G les plus puissants. Dans ces environnements, on a généralement un langage couplé avec une série d'outils tels que des générateurs d'écran permettant de développer des systèmes complexes.

Certains générateurs d'applications ne permettent que de générer un partie de l'application. Ils doivent alors être liés à des langages de programmation ou du moins être capables de faire appel à des routines écrites dans un langage différent. Cependant, les bons générateurs d'application incluent un langage procédural de sorte qu'ils aient peu ou pas besoin de recourir au service des langages de troisième génération.

1.5. Conclusion

Bien qu'il n'existe pas une distinction simple et tranchante qui permette de qualifier un langage de L4G ([Buck90]), nous venons cependant de relever les principales caractéristiques qu'ils présentent généralement. C'est grâce à la présence ou non de ces caractéristiques dans un langage que l'on peut déterminer s'il est qualifiable de langage de quatrième génération.

Chapitre 2

Chapitre 2 : Les outils de développement d'applications

2.1. Introduction

Quand les interfaces graphiques ont commencé à se répandre largement, le besoin d'une nouvelle génération d'environnements de développement est apparu clairement. En effet, l'écriture de programmes supportant un système de fenêtrage (comme Windows ou Motif) est bien plus complexe que son équivalent en mode caractère. Cela nécessite bien souvent un volume de code plus de deux fois supérieur à celui requis par le mode caractère, le tout reposant sur un concept de développement entièrement nouveau : la gestion d'une boucle d'événements ([Lefe93]).

Pour faire face à ce problème, on a assisté à l'apparition d'outils de développement tels que SQLWindows, PowerBuilder, NS-DK, Visual Basic ou encore ObjectView.

Nous allons en donner une description générale, en montrant leurs caractéristiques principales, qui vaut pour les différents outils. Cependant, nous illustrerons parfois notre propos avec un outil particulier qui est PowerBuilder. Si nous avons choisi de privilégier PowerBuilder, c'est parce que cet outil est utilisé dans le quatrième chapitre de ce mémoire.

On peut également noter que tous ces outils ne présentent pas de grandes différences dans leurs principes de fonctionnement et qu'au fur et à mesure où de nouvelles versions sortent, ils ont de plus en plus tendance à offrir les mêmes fonctionnalités (concurrence oblige !).

Nous avons relevé quatre caractéristiques fondamentales de ces outils.

La première caractéristique de ces outils est qu'il permettent de faire ce que l'on appelle de la programmation visuelle. Deuxièmement, avec ces outils, la programmation est événementielle. La troisième caractéristique est que l'accès aux données se fait en mode client-serveur. La dernière caractéristique est que ce sont des outils idéaux pour faire du prototypage.

Nous allons d'abord développer ces quatre caractéristiques après quoi nous montrerons que ces outils constituent bel et bien des L4G.

2.2. Description de ces outils

2.2.1. Programmation visuelle

Avec l'arrivée sur le marché des outils de développement dont on vient de parler ainsi que l'apparition d'autres outils proposant des interfaces interactives graphiques, le terme programmation visuelle est entré dans le lexique informatique.

Le but de la programmation visuelle est de faciliter le travail de développement des applications au programmeur, et, dans une moindre mesure, de permettre à des programmeurs non-professionnels de développer leurs propres applications ([Varh94]).

Les outils de développement sont visuels en ce sens qu'ils permettent de créer l'interface d'une application de façon interactive. Pour ce faire, ils fournissent au développeur une boîte à outil ou un menu contenant les différents objets interactifs qui pourront se trouver dans les différentes fenêtres de son application. Ces objets peuvent être des boutons de commande, des listes de sélection, des champs d'édition, ...

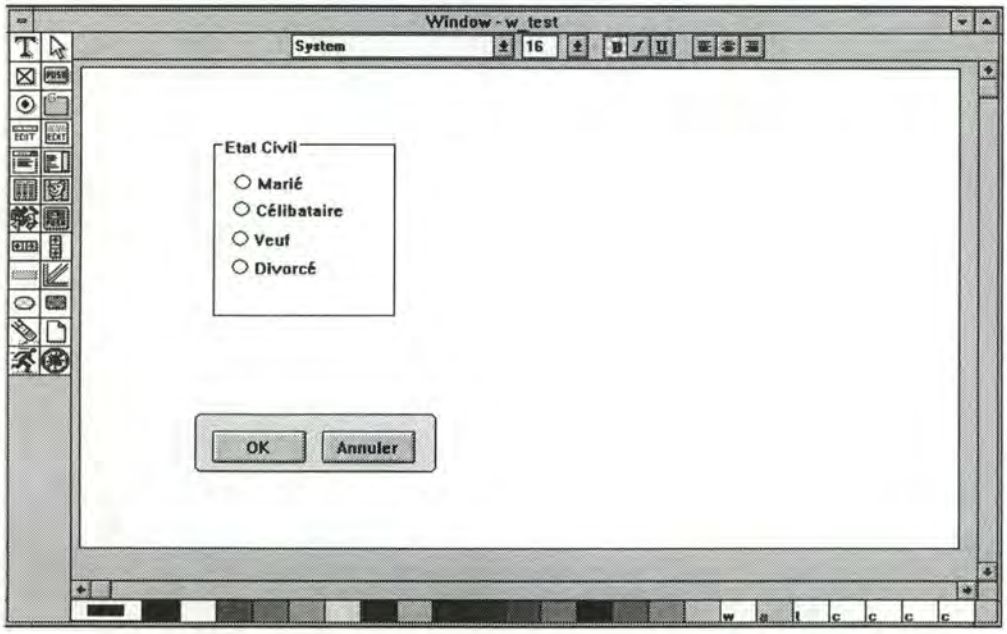


Figure 2.1 : le *Window Painter* de PowerBuilder

Le développeur peut alors aller chercher l'objet qu'il désire et le placer en manipulation directe à l'aide de la souris sur une zone de l'écran qui correspondra à une fenêtre de son application.

La figure 2.1. montre le *Window Painter* de PowerBuilder qui permet de créer une fenêtre de la façon que l'on vient de décrire.

Pour obtenir la fenêtre représentée à la figure 2.1, on a été chercher dans la boîte à outil à gauche de l'écran les différents objets et on les a placés où on le désirait à l'écran, en leur donnant la taille voulue.

Mais outre le fait que l'on puisse aller chercher différents objets interactifs dans une boîte à outil, on peut également très facilement, et toujours de manière visuelle, définir leurs attributs.

La figure 2.2 montre les attributs d'un champ d'édition dans PowerBuilder. Il est très facile, comme on le voit de modifier les attributs que l'on voudra donner à ce champ. Par exemple, si on désire que l'utilisateur ne puisse pas écrire dans ce champ (donc qu'il serve uniquement à afficher des informations), il suffira de cocher la boîte à cocher *Display Only*.

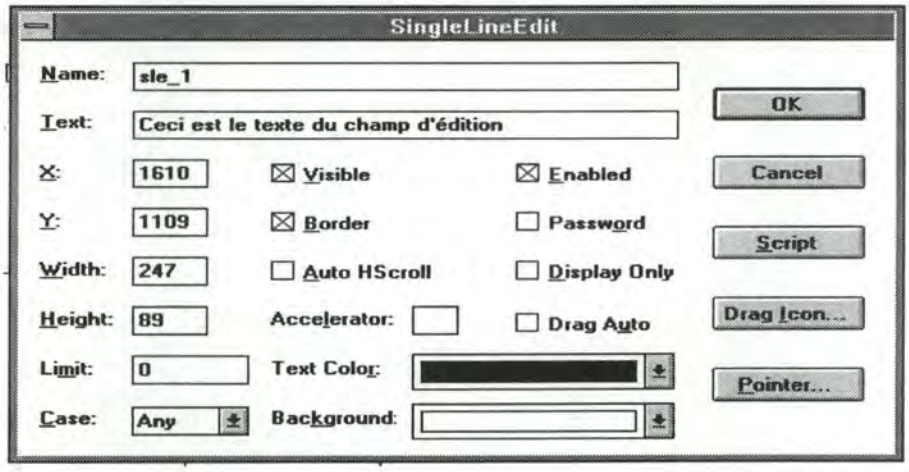


Figure 2.2 : Les attributs d'un champ d'édition

On peut également modifier de la sorte les attributs d'une fenêtre comme le montre la figure 2.3.

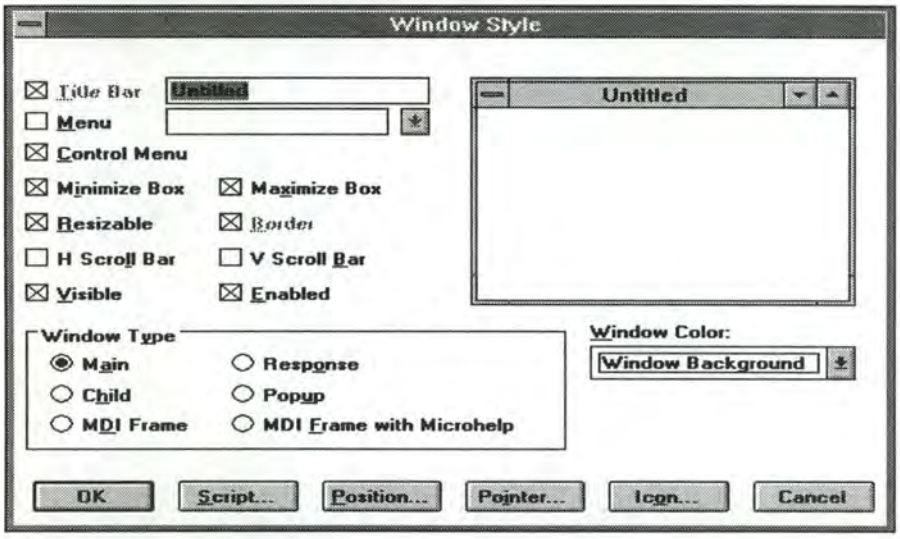


Figure 2.3 : Les attributs d'une fenêtre

On le voit, toute l'interface de l'application pourra être créée très facilement et très rapidement, tout en sachant que le développeur voit exactement ce que verra l'utilisateur final.

On voit immédiatement l'avantage qu'apporte cette création visuelle de l'interface lorsque l'on sait que le développement d'applications ayant une interface utilisateur sophistiquée (et c'est bien notre cas) en n'utilisant pas la programmation visuelle est une tâche complexe qui prend beaucoup de temps. Des études ont d'ailleurs montré que 50 à 80 pour-cent du code de ces applications concerne la partie interface utilisateur ([Lee90]).

Bien évidemment, pour créer une application, il faut plus qu'un ensemble d'objets interactifs. Il faut pouvoir enchaîner les fenêtres et réaliser des traitements sur des données. C'est pourquoi ces outils disposent d'un véritable langage de programmation qui offre d'une part des primitives pour agir sur les

différents objets interactifs (par exemple ouvrir une fenêtre ou modifier un attribut d'un objet quelconque) et d'autre part des primitives permettant de réaliser des opérations plus générales comme par exemple la concaténation de deux chaînes de caractères ou un accès aux données.

A ce niveau encore, certains outils proposent une aide visuelle au développeur. En effet, on retrouve parfois des *browsers* qui permettent d'accéder facilement aux primitives ou aux attributs concernant un objet interactif bien particulier. Ainsi, si l'on veut connaître les fonctions que l'on peut appliquer à un bouton de commande, le *browser* permet de les obtenir très rapidement. Le développeur peut ensuite faire un couper-coller de l'appel de la fonction qui l'intéresse (par exemple la fonction permettant de redimensionner le bouton), ce qui lui évite de devoir l'écrire lui-même.

Nous allons maintenant voir comment les traitements vont être pris en compte dans ces outils.

2.2.2. Programmation événementielle

Avec l'apparition des interfaces utilisateurs graphiques, on est passé d'un ancien type de dialogue contrôlé par l'ordinateur qui présentait à l'utilisateur des écrans successifs, à un type de dialogue contrôlé par les actions de l'utilisateur, traduites sous forme d'événements auxquels l'ordinateur doit répondre. L'interface est devenue événementielle ([Mein91]).

Une des difficultés majeures dans la construction d'interfaces graphiques réside précisément dans le contrôle du dialogue. Simple dans le cas d'un dialogue synchrone tel que questions-réponses ou menus à choix unique dans lesquels peu de possibilité de choix sont laissées à l'utilisateur, il devient complexe dans le cas de l'interface événementielle où les possibilités de choix de l'utilisateur sont considérables.

Le **modèle objet** ou **multi-agents** va permettre de répartir le dialogue sur une multitude d'objets de présentation.

Ce modèle, que nous allons voir, provient du paradigme de l'orienté-objet. Nous n'allons pas, dans ce mémoire revoir les concepts de l'orienté-objet, et le lecteur intéressé pourra se reporter entre autre à [Voss91] pour d'amples informations sur ce paradigme.

Dans le modèle multi-agents, le dialogue est réparti sur les objets ou agents de présentation, également appelés interacteurs ([Mein91]).

Les objets interactifs manipulés dans l'interface apparaissent comme des médiateurs entre le monde abstrait du système et le monde concret de l'utilisateur, avec un double comportement : *comportement interne* vis-à-vis de l'application et *comportement externe* vis-à-vis de l'utilisateur ([Mein91]).

Les événements, actions de l'utilisateur sur les différents objets, se traduisent en messages adressés à ces objets. Ces messages sélectionnant alors les méthodes responsables des comportements internes et externes des objets ([Mein91]).

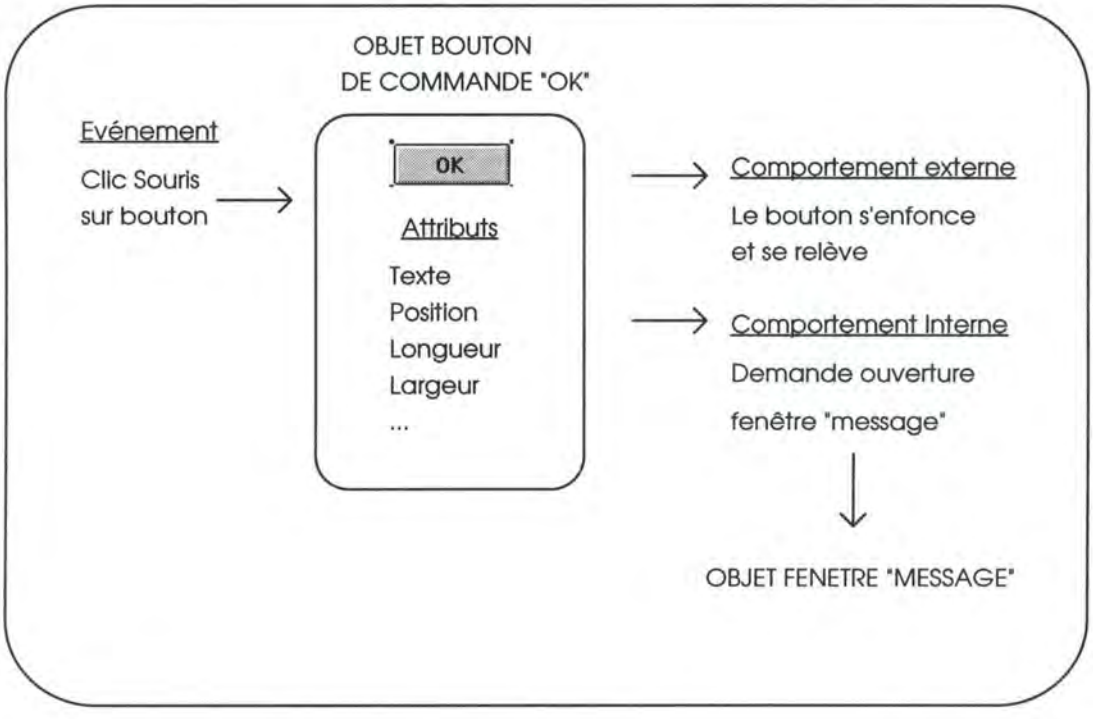


Figure 2.4 : exemple d'agent, le bouton de commande

On peut illustrer ce modèle à travers un exemple qui est représenté à la figure 2.4.

Soit un bouton de commande "OK". On veut que lorsque l'utilisateur clique avec sa souris sur ce bouton, il y ait une fenêtre qui s'ouvre affichant un message à l'utilisateur.

L'événement qui est le clic sur le bouton OK va générer un message qui sera envoyé au bouton de commande. En réaction à ce message, le bouton de commande va d'une part sélectionner la méthode correspondant au comportement externe et d'autre part sélectionner la méthode correspondant au comportement interne.

La méthode externe se chargeant alors de faire en sorte que l'utilisateur ait l'impression que le bouton s'enfonce et se relève, tout comme s'il avait appuyé sur le bouton d'un ascenseur. La méthode interne se chargeant, quant à elle, de demander l'ouverture de la fenêtre en envoyant un message à l'objet fenêtre.

L'objet fenêtre ayant à son tour, en réaction au message reçu, un comportement externe (affichage à l'écran de la fenêtre) et un comportement interne.

Avec les outils de développement dont on est en train de parler, on va développer les applications en respectant le modèle multi-agents.

Le code de l'application ne sera donc pas constitué d'une séquence continue décrivant les étapes à suivre du début à la fin du déroulement de l'application, mais il sera constitué de nombreuses séquences de code répondant chaque fois à un événement bien particulier. C'est ce que l'on appelle la programmation événementielle.

De plus, grâce à ces outils, le développeur ne devra pas se charger de développer les méthodes responsables du comportement externe des différents objets, ni se charger de récupérer et d'analyser les différents événements qui peuvent survenir.

Ainsi, si le développeur place un bouton de commande à l'écran (en utilisant la programmation visuelle comme on l'a vu au point précédent) et qu'il ne fait rien d'autre, lorsqu'il exécutera l'application et qu'il cliquera sur ce bouton, alors ce dernier s'enfoncera et remontera ensuite. La seule chose que doit faire l'utilisateur est de développer une méthode responsable du comportement interne du bouton de commande en réaction au clic sur ce bouton.

Nous allons illustrer cela avec l'outil de développement PowerBuilder. Pour chaque objet interactif, on dispose de la liste des événements par défaut qui peut leur survenir. Dans le cas d'une fenêtre, on aura les événements ouverture fenêtre, fermeture fenêtre, fenêtre activée, fenêtre désactivée, ... Mais les événements seront différents dans le cas du bouton de commande : clic sur le bouton, le bouton reçoit le focus, le bouton perd le focus,...

Le travail du développeur sera de programmer le code en réaction à la survenance des événements auxquels il désire attacher une importance. Il est clair que tous les événements qui surviennent aux différents objets interactifs ne vont pas forcément l'intéresser. Ainsi, il peut très bien ne rien faire lorsqu'une fenêtre s'ouvre et attendre que l'utilisateur agisse sur les objets contenus dans cette fenêtre.

Le développeur utilisant PowerBuilder va mettre le code de son application dans des petites "boîtes" correspondant chaque fois à un événement survenant à un objet bien particulier.

La figure 2.5 montre une "boîte" correspondant au programme qui s'exécutera en réaction au clic sur le bouton de commande *cb_1* (chaque objet est identifié par un nom). Dans ce cas, le clic sur le bouton aura pour effet d'ouvrir la fenêtre *w_message*.

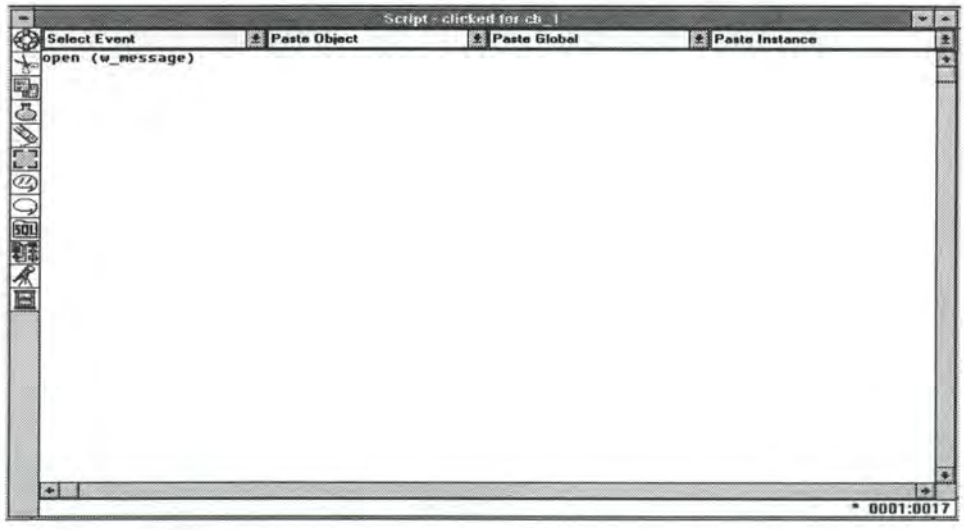


Figure 2.5 : le code en réaction au clic sur un bouton de commande

La structure d'un programme se trouvant enfermé dans une "boîte" sera quant à elle proche de celle des langages de troisième génération. On aura la possibilité d'avoir des boucles, il faudra déclarer des variables, ... De plus le langage permet en général de faire ce que l'on pouvait faire avec un langage de troisième génération (traitement de chaînes de caractères, conversion de types,...).

Comme nous le verrons au point suivant, ces outils permettent aussi à ce niveau d'utiliser un langage SQL afin d'accéder aux données.

Il est clair qu'avec ce genre d'outil, le développement d'une application est pensé entièrement en fonction de l'interface puisque toute action sera toujours réalisée en réaction à un événement survenu à un objet de présentation.

2.2.3. Accès aux données en mode Client-Serveur

Avec ces outils, l'accès aux données se fait en mode client-serveur. Ils offrent bien quelques fonctions de manipulation de fichier, mais la grande puissance de ces outils est qu'ils permettent d'accéder facilement à des données se trouvant sur un serveur de données.

On peut définir une architecture client-serveur comme étant un cas particulier de traitement coopératif ([Bers92], [Lefe93]) où on a une application qui est partagée entre un système client et un système serveur ([Bers92]). Le client communiquant avec le serveur en lui passant des requêtes, généralement à travers un réseau. Le serveur répondant alors à ces requêtes, soit sous la forme d'informations spécifiques correspondant à la demande du client, ou soit sous la forme d'un message signalant que la requête a bien été effectuée (ou qu'elle a échoué) ([Gutt93]).

En fait, il existe trois grands types de client-serveur. Il y a le client-serveur de données, le client-serveur de présentation et le client-serveur de procédure ([Lefe93]).

Nous n'allons pas détailler dans ce mémoire le modèle client-serveur et le lecteur intéressé trouvera plus d'information sur ce modèle dans [Lefe93] ou encore dans [Bers92].

En bref, nous dirons que l'on se trouve face à du **client-serveur de présentation** lorsque l'on déporte la gestion de l'affichage sur un serveur

spécialisé (appelé serveur d'affichage ou encore "serveur X"). On a alors sur ce serveur ce que l'on appelle un gestionnaire de fenêtre (typiquement X-Window).

Concrètement, l'application cliente envoie des requêtes de demande d'affichage d'objets (avec en paramètre les caractéristiques de l'objet : taille, emplacement, ...) vers le serveur X. Le serveur traite alors ces requêtes et affiche les objets en fonction des paramètres reçus (grâce au gestionnaire de fenêtre). Il est ensuite capable de gérer de façon indépendante les événements qui ne sont pas significatifs pour l'application cliente (par exemple, le redimensionnement d'une fenêtre). Lorsque survient un événement qui est considéré comme significatif (par exemple le choix d'un item de menu par l'utilisateur), il est transmis à l'application cliente sous forme d'un message. La réception du message avertissant l'application cliente de l'occurrence de tel ou tel événement, ce qui entraîne l'émission d'une nouvelle requête en réaction à cet événement.

Avec le client-serveur de présentation, il faut être bien conscient que la machine que l'utilisateur a devant lui n'est pas le client, mais bien le serveur.

Avec le **client-serveur de procédure**, on a une application cliente qui demande l'exécution d'une procédure stockée sur un serveur. L'exécution de cette procédure par le serveur (qui contient un module d'exécution des procédures) pouvant entraîner à son tour une ou plusieurs requêtes vers un serveur de données.

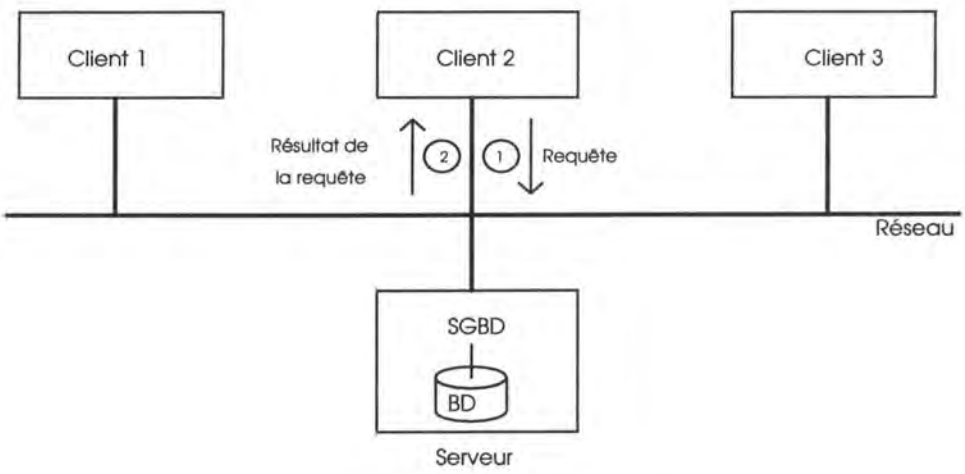


Figure 2.6 : le client-serveur de données

En ce qui concerne le **client-serveur de données**, il s'agit du type de client-serveur qui est le plus fréquemment utilisé, et le plus souvent, lorsqu'on parle de client-serveur, on entend client-serveur de données. C'est aussi le type de client-serveur qui va directement nous intéresser puisque c'est ce type qui est mis en oeuvre grâce aux outils de développement dont nous parlons.

Avec le client-serveur de données, comme le montre la figure 2.6, le système de gestion de bases de données (SGBD) se trouve sur le serveur de données. Les postes clients du réseau ne disposent pas de SGBD local qui leur permettrait de gérer les données.

Lorsqu'une application cliente désire accéder aux données, elle transmet une requête au SGBD à travers le réseau (le plus souvent, il s'agira d'une requête SQL). Ensuite, lorsque le SGBD reçoit la requête, il la traite (tout en veillant à conserver l'intégrité des données) et renvoie le résultat (et uniquement le résultat) à l'application cliente. La charge d'informations circulant sur le réseau sera donc réduite puisque que l'on fera transiter uniquement le résultat de la requête, et non toutes les données nécessaires à l'obtention du résultat de la requête.

A l'heure actuelle, les principaux SGBD relationnels que l'on trouve sur le marché (Oracle, Sybase, SQLBase, Ingres,...) permettent de faire du client-serveur de données.

Les outils de développement dont on parle permettent de développer rapidement une application cliente qui accède aux données se trouvant sur un serveur de données.

En général, ces outils proposent dans leur langage un sous-langage permettant d'accéder aux données. Il s'agira généralement d'un langage SQL relativement indépendant (appelons le langage propriétaire) par rapport aux différentes sortes de SQL que l'on rencontre sur le marché (en effet, chaque éditeur de SGBD propose son propre langage SQL avec ses propres particularités).

Les outils de développement proposent alors des interfaces avec les principaux SGBD du marché. Ce sont ces interfaces qui vont transformer automatiquement les commandes écrites dans le langage propriétaire en commande conforme au SQL du SGBD bien particulier sur lequel on veut travailler.

On constate que grâce à cette façon de fonctionner, une application écrite pour accéder au SGBD Sybase pourra accéder au SGBD Oracle sans que le programmeur n'ait rien à changer dans son programme. La seule chose à faire sera d'utiliser l'interface d'Oracle et non celle de Sybase.

Cependant, on remarque qu'avec cette technique, d'une part, on ne peut accéder qu'aux SGBD pour lesquels l'outil de développement fournit une interface, et que d'autre part, pour un même SGBD, il faudra autant d'interfaces qu'il y a d'outils de développement.

Microsoft propose une solution qui permet de faire face à ce problème : **ODBC** (*Open DataBase Connectivity*).

En créant ODBC , Microsoft a voulu offrir un moyen d'accéder aux SGBD qui soit puissant, ouvert et surtout qui soit indépendant de l'éditeur du SGBD ([Micr92]).

Le principe de fonctionnement d'ODBC est relativement simple, et on peut faire le parallèle avec la gestion des imprimantes telle qu'elle est prise en compte dans Windows 3 ([Lefe93]).

En effet, avant l'apparition de Windows 3, chaque application devait gérer de façon propre son accès aux imprimantes. C'est à dire qu'avec un certain traitement de texte, on vous fournissait toute une collection de programmes pilotes (*driver*) pour les principales imprimantes que l'on trouvait sur le marché. Le pilote constituant l'interface entre l'application et l'imprimante. L'utilisateur installait alors le pilote correspondant à son imprimante (encore fallait-il qu'il en existe un !), et il lui restait à espérer que ce pilote ait été proprement réalisé (puisque les pilotes n'étaient pas réalisés par les fabriquants des imprimantes, mais par les éditeurs de logiciels).

Avec l'apparition de Windows 3, c'est tout à fait différent. Les développeurs écrivent leurs applications en faisant abstraction de l'imprimante qui sera utilisée. Pour cela ils utilisent des fonctions proposées par un gestionnaire d'impression intégré à Windows. Windows fournissant alors un et un seul pilote pour chaque imprimante. On n'a donc plus, pour une même imprimante autant de pilotes qu'il n'y a d'applications. Le gestionnaire d'impression utilisant alors le pilote choisi par l'utilisateur afin de réaliser une impression correcte.

De plus, maintenant, le pilote est développé par le constructeur de l'imprimante, ce qui doit en assurer une meilleure qualité.

Pour ODBC, le principe est tout à fait le même, si ce n'est que les imprimantes ont été remplacées par des SGBD.

Avant ODBC, lorsqu'une application cliente voulait accéder à un SGBD, il fallait utiliser une interface offerte par le SGBD. Cette interface, qui se trouvait au niveau du poste client, fournissait un ensemble de fonctions d'accès au SGBD (par exemple, la connexion avec une base de donnée).

La figure 2.7 montre l'architecture d'une application utilisant ODBC ([Micr92], [Lefe93]).

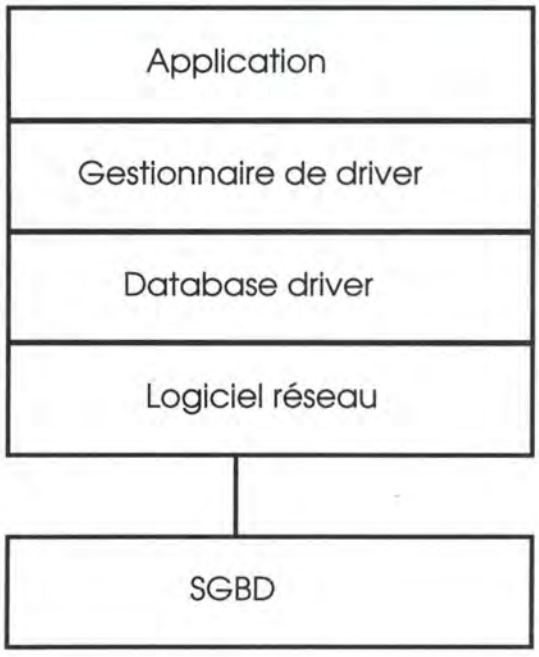


Figure 2.7 : l'architecture d'ODBC

Avec ODBC, on n'utilise plus une interface propre à tel ou tel SGBD, mais on utilise une interface universelle propre à ODBC.

Concrètement, les applications utilisent maintenant le *gestionnaire de driver* que propose ODBC et qui offre des fonctions normalisées d'accès aux données permettant d'exprimer des requêtes dans une grammaire SQL standard.

Les requêtes sont ensuite passées au *database driver* qui va se charger de traduire ces requêtes standards en requêtes compréhensibles par le SGBD cible. La requête est alors envoyée vers le SGBD à travers le réseau en utilisant un logiciel réseau, qui cette fois, est propre au SGBD auquel on veut accéder.

Avec ce système tout éditeur de SGBD qui voudra être accessible grâce à ODBC n'a qu'à développer son propre driver. A l'heure actuelle, il existe des drivers pour Oracle, Sybase, dBase, Paradox, Ingres, Informix, ...

En ce qui concerne les outils de développement, ils proposent généralement une interface avec ODBC. Cela ne change absolument rien au principe de fonctionnement sans ODBC, mais cette fois, les éditeurs d'outils de développement n'ont plus besoin de développer telle ou telle interface pour les différents SGBD du marché.

2.2.4. Outils de développement et prototypage

Le cycle de vie traditionnel de développement d'applications représenté à la figure 2.8 a été utilisé pendant des dizaines d'années et il est encore utilisé à l'heure actuelle ([Keye92]).

Cependant, comme nous allons le voir, ce cycle de vie pose des problèmes pour le développement d'applications interactives (ce qui constitue de plus en plus la majorité des applications).

Selon certains auteurs, entre 70 ([Wils88]) et 80 % ([Keye92]) des budgets informatiques sont utilisés pour faire de la maintenance plutôt que pour développer de nouveaux produits.

La maintenance consiste à corriger les erreurs et à adapter les programmes pour répondre aux nouveaux besoins ([Wils88]).

Il est très fréquent de rencontrer des utilisateurs qui, après avoir attendu des mois ou même des années pour que leur projet soit terminé, demandent immédiatement des modifications du nouveau système. Ces demandes de

modifications sont le résultat direct d'un cycle de développement des applications qui est souvent inflexible et qui traîne en longueur ([Keye92], [Mart85]).

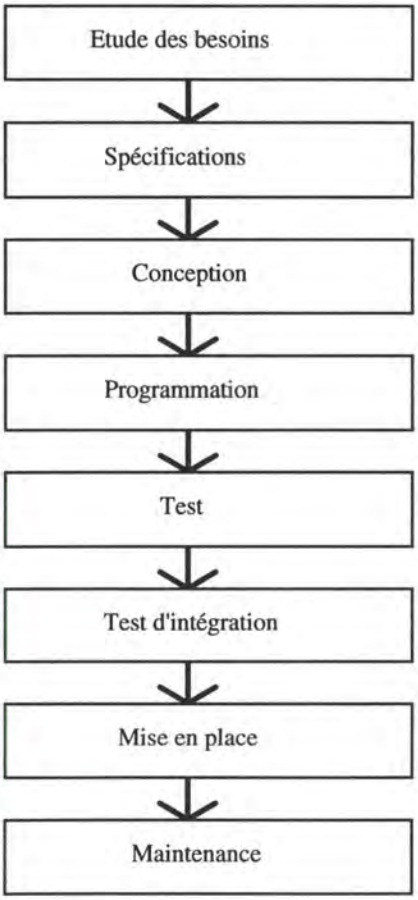


Figure 2.8 : Cycle de vie traditionnel de développement des applications

Avec le cycle de développement traditionnel, bien que l'on sollicite l'utilisateur final, il n'y a pas réellement moyen pour ce dernier de voir et de travailler avec un modèle du produit final avant que le projet ne soit déjà loin dans sa réalisation.

De ce fait, lorsque l'on commet des erreurs durant la phase de spécifications, elles restent souvent cachées jusqu'à ce que l'on passe à la phase de

programmation. A ce moment, il est souvent trop tard pour essayer de réparer les erreurs, et les utilisateurs finaux doivent se résigner à attendre l'étape de maintenance pour faire part de leurs problèmes ([Keye92]).

D'autre part, il est important de noter également que des spécifications incorrectes ne sont pas les seuls problèmes qui peuvent survenir. Parfois, il arrive tout simplement que les utilisateurs finaux ne sachent pas réellement ce qu'ils veulent tant qu'ils ne le voient pas ([Boeh88], [Keye92], [Mart85]).

On se retrouve face à des situations où les utilisateurs disent : *"Je ne peux pas vous dire ce que je veux, mais je le saurai quand je le verrai"* ([Boeh88]).

On le voit, le cycle de développement traditionnel n'est pas adapté pour les applications interactives où il est vital que l'application soit bien acceptée par l'utilisateur sous peine de ne pas être utilisée ou à tout le moins d'être sous utilisée.

Il est donc nécessaire d'impliquer beaucoup plus l'utilisateur tout au long du développement de l'application.

Une solution permettant de résoudre ce problème est de travailler avec des **prototypes** que l'on crée rapidement et que l'on présente aux utilisateurs pour voir si ils les satisfont. Ces prototypes devant être rapidement modifiables afin de pouvoir s'adapter aux nouveaux désirs des utilisateurs.

Selon [Mein91], un prototype peut être construit à différents niveaux :

- affichage d'écran seulement

- simulation écran/clavier/souris

- fonctionnalités limités

- fonctionnalités complètes, performances limitées

- vérification de performance

Du fait que les outils de développement d'applications graphiques, et tous les L4G en général, ont ce caractère visuel dont on a parlé et permettent de

développer rapidement une application, ils sont des outils qui se prêtent parfaitement au prototypage ([Varh94], [Keye92]).

Il nous semble qu'un prototype développé avec ces outils doit répondre aux trois premiers niveaux de [Mein91].

En effet, puisque l'on développe avec ces outils des applications interactives présentant une interface graphique, il est primordial que le prototype donne une bonne idée de ce que sera effectivement l'interface du produit fini (autant les écrans que l'utilisation de la souris).

De plus, il paraît essentiel que le prototype *présente* les différentes fonctionnalités (même si elles ne sont pas entièrement implémentées) qu'offrira le produit final.

Par exemple, si on veut montrer qu'il y aura une fonctionnalité qui devra permettre d'insérer un nouvel élément dans un tableau, alors on peut déjà mettre, dans le prototype le bouton de commande correspondant à l'exécution de cette action. Il ne sera pas nécessaire d'implémenter cette fonction au niveau du prototype puisque la présentation du bouton de commande devrait suffire à l'utilisateur pour comprendre l'action qu'il pourra réaliser.

On pourrait également faire apparaître une fonctionnalité dans le prototype en plaçant un item de menu qui correspond à la fonction souhaitée. On pourrait alors avoir, lorsque l'utilisateur choisit cet item, l'affichage à l'écran d'un message expliquant, dans la langue de l'utilisateur, ce que réalisera cette fonction.

Tout cela, afin que l'utilisateur puisse se rendre compte des fonctionnalités supplémentaires dont il aimerait disposer et qu'il ne retrouve pas dans le prototype, ou encore afin qu'il puisse signaler que certaines fonctionnalités prévues ne l'intéresse pas, et que ce n'est donc pas la peine de les développer.

Avec les outils de développement dont on parle, il arrive que le développeur et l'utilisateur final s'asseyent côte à côte devant l'écran de l'ordinateur et développent rapidement un prototype de l'application. Le prototype étant ajusté directement par le développeur pour faire face aux désirs de l'utilisateurs ([Keye92], [Lint94]).

Mais il est clair qu'il n'est pas toujours possible de mettre côte à côte devant un écran le développeur et l'utilisateur final. Dans ce cas, le développeur réalise

rapidement un prototype de l'application en fonction des spécifications dont il dispose. Ce prototype, bien que ne présentant que des fonctionnalités limitées, doit permettre à l'utilisateur final de se faire une bonne idée de ce que sera le produit fini.

Ensuite, le prototype est remis à l'utilisateur qui l'utilise et qui découvre ainsi ses nouveaux besoins. On obtient alors de nouvelles spécifications qui sont traitées par le développeur. On peut continuer ainsi de suite jusqu'au moment où le prototype satisfait l'utilisateur final.

On se retrouve donc face à un processus itératif où le développeur et l'utilisateur final raffinent continuellement le prototype jusqu'à aboutir à un prototype qui satisfait l'utilisateur ([Keye92]).

Il restera ensuite au développeur à implémenter pleinement les différentes fonctionnalités prévues dans le prototype.

Cependant, il peut arriver, lorsqu'on développe des applications relativement simples que le prototype auquel on est aboutit corresponde au produit final ([Keye92], [Mart85]).

2.3 En quoi ces outils constituent-ils des L4G

Nous allons maintenant voir que ces outils de développement constituent bel et bien des langages de quatrième génération. Pour ce faire, nous allons passer en revue les différentes caractéristiques des L4G que nous avons relevées dans le chapitre précédent et nous montrerons en quoi ces caractéristiques se retrouvent dans les outils de développement.

2.3.1. Langage non-procédural

Rappelons qu'un langage procédural spécifie **comment** accomplir quelque chose, tandis qu'un langage non-procédural spécifie **quoi** accomplir, sans décrire comment ([Mart85]).

Les outils de développement ne sont pas des outils qui sont entièrement non-procéduraux, mais des outils qui combinent à la fois les avantages de la programmation procédurale et ceux de la programmation non-procédurale.

Ainsi, l'interface est construite visuellement de façon non-procédurale. Le développeur se contente de dessiner directement à l'écran ce qu'il veut obtenir dans son application, il ne se préoccupe pas de savoir comment cela doit être implémenté concrètement.

De plus, on a vu que les outils de développement offrent généralement un langage d'accès au données non-procédural de type SQL.

Donc, tant au niveau de la création de l'interface qu'au niveau de l'accès aux données, ces outils offrent de grandes possibilités de programmation non-procédurale.

Par contre, en ce qui concerne le développement des parties de code qui réagissent aux divers événements survenant aux objets interactifs, il est réalisé dans un langage procédural de type langage de troisième génération. Par exemple, pour Visual Basic, ce langage procédural sera le Basic.

2.3.2. Fonctionnalités limitées

On a vu que les L4G varient beaucoup dans leur puissance et dans leur capacité, ils vont du générateur de rapport au générateur d'applications complètes.

En ce qui concerne les outils de développement, il s'agit d'outils permettant de développer des applications complètes. Cependant, il faut reconnaître que ces outils ne sont adaptés que pour des applications **de gestion** qui sont hautement **interactives**.

2.3.3. Simplicité d'utilisation

Il est clair que, puisque ces outils offrent la programmation visuelle non-procédurale, ils seront relativement simple à utiliser, du moins en ce qui concerne la création de l'interface.

Ils proposent également une grande simplicité dans l'accès aux données puisque, là encore, il est en général réalisé en utilisant un langage de quatrième génération tel que SQL.

Mais ils offrent généralement d'autres caractéristiques qui en simplifient fortement l'utilisation telles que des *browsers* ou encore une aide en ligne bien faite.

Cependant, on ne peut pas réellement dire que ces outils soient proches des utilisateurs (*user-friendly*). Il faut bien être conscient qu'en dehors de la création de l'interface où il faut plutôt avoir des qualités d'ergonome que de programmeur, tous les autres aspects du développement demandent quand même de bonnes habitudes de programmation. Il faudra en effet utiliser un langage où l'on doit déclarer des variables, utiliser des boucles, faire appel à des fonctions,...

Cela dit, notre expérience personnelle dans l'utilisation de ces outils nous a montré qu'en ayant les connaissances de programmation de base, ont devient beaucoup plus vite opérationnel et productif qu'avec un langage de troisième génération.

2.3.4. Facilité de débogage

Les outils de développement dont on parle offrent en général un débogueur relativement simple qui est intégré à l'environnement de développement. Le débogage est donc effectué au niveau de l'outil de développement et non au niveau du code objet.

De plus, dans ces environnements, le processus de débogage est généralement réalisé en utilisant la pleine potentialité du mode graphique (fenêtrage, utilisation de la souris,...) ce qui rend les débogueurs beaucoup plus simples d'utilisation.

2.3.5. Facilité de maintenance

Une application développée avec un de ces outils peut être reprise et modifiée facilement au niveau de l'environnement de développement.

De plus, il nous semble que les caractères de programmation visuelle et événementielle qu'offrent ces outils permettent à un développeur de pouvoir plus facilement comprendre et modifier une application développée par une autre personne.

En effet, en ce qui concerne l'interface, on a vu qu'elle apparaissait de la même façon au développeur et à l'utilisateur final. Le développeur n'aura donc aucun effort à faire pour comprendre et modifier l'interface d'une application développée par un autre.

En ce qui concerne le code de l'application, on a vu qu'il était constitué de séquences de code qui sont écrites en réaction aux événements survenant aux objets interactifs. Le développeur pourra donc se focaliser uniquement sur la compréhension du code qui l'intéresse, il n'aura pas besoin d'avoir une compréhension générale du programme pour pouvoir modifier l'application.

Finalement, on a déjà dit que les outils de développement sont des outils qui se prêtent bien au prototypage. On a également vu que le prototypage doit permettre d'aboutir au développement d'applications qui satisfont au mieux les besoins de l'utilisateur.

Dès lors, si l'application satisfait mieux aux besoins de l'utilisateur, les efforts de maintenance qui consistent souvent à adapter l'application aux demandes de l'utilisateur pourront être réduits fortement.

2.3.6. Facilité de la manipulation des données

Comme on l'a dit, ces outils offrent un langage d'accès aux données se trouvant sur un SGBD (souvent relationnel). Ce langage est généralement de type SQL, ce qui signifie que pour le programmeur, les données vont apparaître sous la forme logique de tables sur lesquelles il pourra facilement travailler en connaissant un minimum de commandes.

Nous allons maintenant décrire le concept de *datawindow* qui est la propriété de PowerSoft, l'éditeur de PowerBuilder et qui constitue une interface simple entre une application et un SGBD.

Il s'agit d'un objet intelligent que l'on lie directement à une vue de la base de données (table ou jointure de tables) et qui offre une série de fonctions facilitant grandement la manipulation de cette vue.

Pratiquement, pour utiliser une datawindow, il faut tout d'abord donner une définition d'un **objet datawindow** dans PowerBuilder. Cette définition est constituée de deux parties. D'une part, il s'agira de dire sur quelle vue de la base

de données on veut travailler et d'autre part quel sera le format d'affichage à l'écran de cette vue (tableau, formulaire ou grille).

La figure 2.9 montre un exemple d'une telle définition. Pour donner cette définition, l'utilisateur n'a pas besoin de connaître le langage SQL, tout va se faire de manière visuelle. Il dispose à l'écran des tables qu'il a choisies et à l'aide de la souris, il sélectionne certaines colonnes de tables (dans ce cas les colonnes *nom_cli* et *adresse_cli* de la table *client* et les colonnes *date_com* et *montant_com* de la table *commande*). De même, il pourra exprimer sur quelle colonne va se réaliser la jointure (dans ce cas sur la colonne *num_cli*).

Toujours de manière visuelle, l'utilisateur pourra en fait définir la vue de sa datawindow en donnant les même critères que ceux offerts par le langage SQL (Where, Group by, Sort, ...).

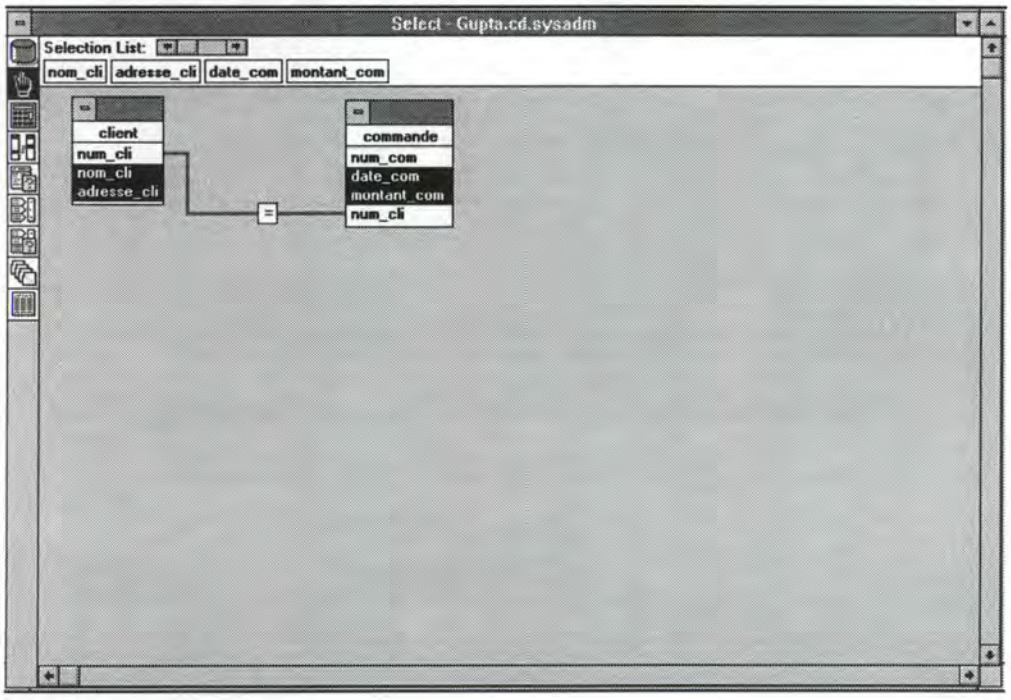


Figure 2.9 : la définition d'une datawindow

Il faut ensuite insérer dans la fenêtre où l'on veut utiliser cet objet datawindow ce que l'on appelle un **datawindow *control***. Il s'agit d'un objet interactif que l'on crée de la même façon que l'on crée un bouton de commande ou une liste de sélection, c'est à dire en utilisant la boîte à outil qui présente les différents objets interactifs.

Une fois que l'on a placé à l'écran le datawindow *control*, on le lie à un objet datawindow existant. A noter que l'on peut lier un datawindow *control* présent dans une fenêtre à n'importe quel objet datawindow.

La figure 2.10 montre un exemple d'application constituée d'une fenêtre dans laquelle on a inséré un datawindow *control* et on a lié ce *control* à l'objet datawindow que l'on a défini à la figure 2.9. Il faut ajouter que l'on avait choisi, lors de la définition de l'objet datawindow un format d'affichage en grille.

On constate donc qu'automatiquement, les données ont été obtenues dans la base de données sur base de la définition de la datawindow et ces données ont été présentées sous forme d'une grille.

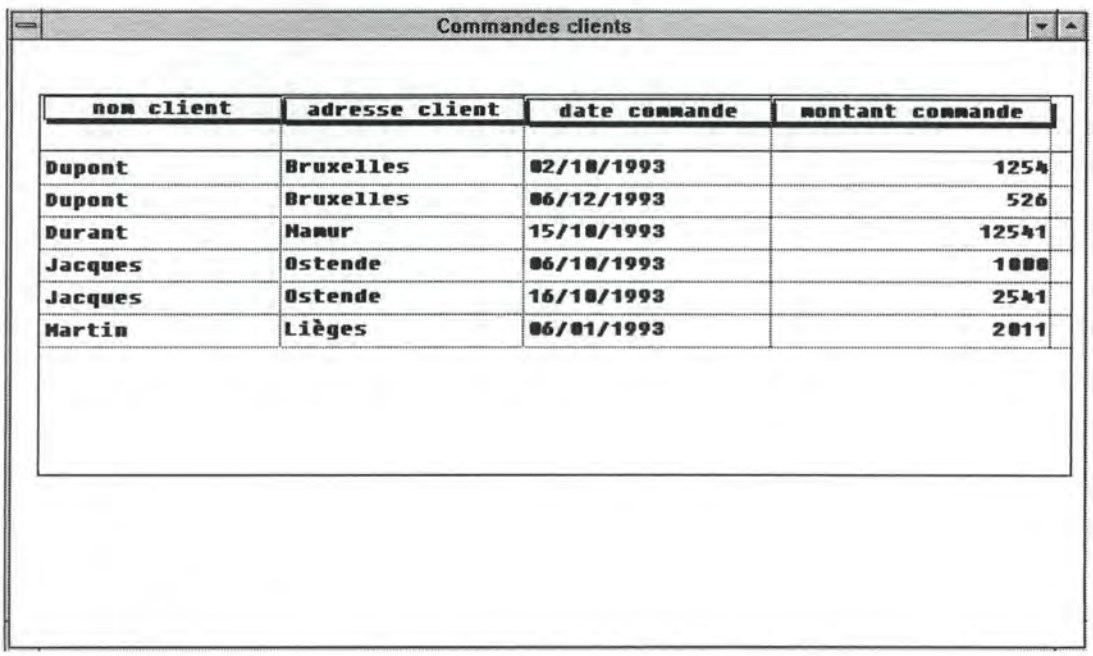


Figure 2.10 : une fenêtre contenant un objet datawindow

PowerBuilder offre alors un grand nombre de fonctions permettant de travailler sur un datawindow *control*. On aura entre autre la possibilité d'insérer de nouvelles données, de faire des modifications de données, de supprimer des données ou encore de mettre à jour la base de données si la datawindow a été modifiée.

Il est encore important de remarquer que les fonctions s'appliquent sur le datawindow *control*, indépendamment de l'objet datawindow. On voit donc l'avantage que cette technique apporte au niveau de la réutilisation. En effet, si on travaille sur un objet datawindow bien particulier à travers un datawindow *control* et aux fonctions que l'on peut appliquer sur lui, il suffit de lier ce *control* à un autre objet datawindow pour que toutes les fonctions qui ont été implémentées sur ce *control* restent valables pour le nouvel objet datawindow.

On vient donc de voir, à travers ce concept de datawindow, un moyen simple et rapide de manipuler des données.

On peut également ajouter que ce concept de datawindow commence à se retrouver dans d'autres outils de développement.

2.3.7. Intégration des aspects interface, traitements et données

Il est évident, après tout ce que l'on vient de dire sur ces outils de développement que les aspects interface, traitements et données sont intégrés dans un seul et même environnement. Une application complète peut être réalisée sans avoir à sortir de l'environnement de travail.

2.4. Conclusion

Nous venons de donner une description des outils de développement du style PowerBuilder, NS-DK, Visual Basic, SQLWindows,... Nous avons vu leurs quatre caractéristiques principales, à savoir qu'ils permettent de faire de la programmation visuelle et événementielle, que l'accès aux données se fait en mode client-serveur et qu'ils se prêtent bien au prototypage.

Nous avons ensuite passé en revue les différentes caractéristiques des langages de quatrième génération et nous avons montré que les outils de développement

présentent bien ces caractéristiques. Nous pouvons donc affirmer que ces outils de développement constituent bel et bien des L4G.

Chapitre 3

Chapitre 3 : Description théorique d'un framework

3.1. Introduction

Maintenant que le génie logiciel est parvenu à maturité, on doit se rendre à l'évidence : on ne parviendra à diminuer les coûts de production et de maintenance d'un logiciel, tout en augmentant sa qualité, qu'en développant une politique de réutilisation ([Somm92]).

En effet, on reconnaît depuis longtemps qu'une des faiblesses fondamentales dans le développement de logiciels est le fait qu'un système entièrement nouveau est habituellement construit "à partir de zéro" ([Horo84]).

Nous allons maintenant voir ce qu'on entend par réutilisation après quoi nous verrons le concept de framework permettant de mettre en oeuvre la réutilisation.

3.2. La réutilisation

3.2.1. Deux modes de réutilisation

Selon [Bigg87], les technologies qui sont appliquées au problème de réutilisation peuvent être partagées en deux groupes importants selon la nature des composants réutilisés. On a d'une part les technologies de **composition** et d'autre part les technologies de **génération**.

Dans les technologies de composition, les composants devant être réutilisés sont idéalement inchangés au cours de leur réutilisation. Cependant, l'idéal n'est pas toujours atteint et il arrive que les composants soient modifiés afin de mieux répondre au besoins de la personne qui va les réutiliser.

Dès lors, produire une nouvelle application consiste à appliquer un ensemble de principes de compositions bien définis pour connecter les composants. On peut citer comme exemples des composants tels que les fonctions, les programmes ou encore les objets.

En ce qui concerne les technologies de génération, elles ne sont pas aussi faciles à caractériser que les technologies de composition. En général, les composants réutilisables sont des modèles pour la génération de code ou des

règles de transformation faisant partie du générateur. La réutilisation consistant alors à faire fonctionner le générateur.

3.2.2. La réutilisation et le paradigme de l'orienté-objet

Avec le développement du paradigme orienté-objet on a mis beaucoup d'espoir dans les technologies de composition pour promouvoir la réutilisation, cela grâce aux caractéristiques principales de l'orienté-objet que sont l'héritage, le polymorphisme, l'encapsulation, ... ([Beck93], [Voss91], [Wirf90]).

Par exemple, lorsque l'on utilise un langage qui gère l'héritage, on peut définir une classe de base qui ne fournisse que les fonctionnalités de base. Lorsqu'on a besoin de fonctionnalités supplémentaires, on crée une nouvelle version qui hérite de cette classe. On n'a alors pas besoin d'implémenter à nouveau les fonctions de la classe de base, on ne fait que les réutiliser.

Il faut remarquer que jusqu'à présent, on s'est principalement attaché au paradigme orienté-objet comme étant un moyen de décomposer de grandes applications en un ensemble d'objets qui interagissent plutôt qu'un moyen de composer de grandes applications à partir de composants réutilisables ([Beck93]).

3.2.3. Nécessité de composants génériques

Malheureusement, il y a peu de chances pour que des composants créés pour une application bien particulière soient réutilisables immédiatement afin de créer une autre application. Il est clair que ces composants ont été conçus en fonction des besoins de l'application pour laquelle ils ont été développés à l'origine.

Une solution qui permettrait de pouvoir réutiliser efficacement des composants serait qu'ils soient généralisés afin qu'ils répondent à une plus grande catégorie de besoins.

C'est à dire qu'on ne développerait plus des composants particuliers satisfaisant une application bien précise, mais on développerait des composants génériques qui pourraient être utilisés pour développer une série d'applications qui se situent cette fois dans un domaine particulier.

Comme on le voit, un composant générique doit être représentatif d'une abstraction d'un domaine d'application bien particulier. Il est donc nécessaire, pour développer ce composant, d'avoir une bonne connaissance du domaine d'application pour lequel le composant est conçu.

Mais développer des composants génériques revient plus cher que de développer des composants dans un but donné, ce qui augmente le coût des projets.

Le choix de développer une politique de réutilisation de composants est une décision qui relève de la politique de l'organisation. En effet, il faut être prêt à augmenter les coûts à court terme en vue de dégager des bénéfices à long terme ([Somm92]).

Voyons maintenant quels sont les avantages d'une telle politique de réutilisation.

3.2.4. Les avantages de la réutilisation

Les avantages que l'on peut retirer d'une réutilisation de composants ne se situent pas uniquement au niveau de la diminution des coûts de développement. En effet, selon [Somm92], la réutilisation permet aussi :

(1) d'augmenter la fiabilité du système en réutilisant des composants qui ont été testés à fond, et plus précisément qui ont fonctionné sur des systèmes opérationnels.

(2) de réduire les risques d'un projet puisque le coût de réutilisation est plus facile à estimer que le coût de développement.

(3) d'utiliser efficacement des spécialistes à qui on demande de développer des composantes réutilisables qui encapsulent leur connaissance, plutôt que de leur faire faire la même chose sur différents projets.

(4) d'implémenter des standards par le biais des composants réutilisables.

(5) de réduire le temps de développement du logiciel puisque la réutilisation de composants permet d'accélérer la production du système et de réduire le temps de validation.

Nous allons maintenant voir comment mettre en oeuvre de façon efficace la réutilisation grâce au concept de framework.

3.3. Le concept de framework

Nous allons en fait reprendre le concept de framework tel qu'il a été développé dans [Beck93].

Dans cette thèse, le concept de framework a largement été décrit en vue de permettre le développement rapide de Systèmes d'Information d'Aide à la Décision (SIAD).

Un SIAD étant un système informatique ayant pour but d'améliorer l'efficacité de la prise de décision, en aidant les décideurs qui utilise des modèles et des données pour résoudre des problèmes de décision semi-structurés ou non-structurés ([Beck93]).

Cependant, nous allons utiliser ce concept de framework pour le développement d'applications en général.

3.3.1. Définition d'un framework

Selon [Beck93], un framework représente une solution générique pour une classe de problèmes donnée, généralement appartenant au même domaine; cette solution pouvant être personnalisée pour répondre à un problème particulier.

En fait, un framework peut être vu comme une architecture d'application de haut niveau qui est constituée d'un ensemble de classes qui sont spécialement conçues pour être affinées et être utilisées comme étant un groupe ([Wirf90]).

Un framework décrit principalement comment les différentes classes qui le composent coopèrent afin d'accomplir l'objectif général de l'application ([Beck93]).

Bien que du code puisse être fourni dans les classes constituant le framework, le rôle essentiel d'un framework est de décrire la façon dont un système est partitionné en composants ainsi que la façon dont ils interagissent. De cette

manière, l'utilisateur d'un framework peut se concentrer sur l'affinage et la combinaison de composants ([Beck93]).

3.3.2. Vers un nouveau cycle de vie dans le développement d'application

La réutilisation de framework afin de développer des applications induit, au vu de la définition que nous avons donné d'un framework, un nouveau cycle de vie dans le développement d'application.

Dans ce nouveau cycle, on peut distinguer deux rôles qui seront généralement joués par des personnes différentes.

En effet, on a dit qu'un framework est une solution générique qui doit être personnalisée afin d'obtenir une application spécifique.

Il est donc clair qu'il faudra d'une part développer le framework en tant que tel, après quoi seulement il pourra être personnalisé.

On voit donc clairement l'apparition de deux rôles correspondant à deux activités différentes. On a d'une part l'ingénieur d'application (*application engineer*) qui est la personne chargée de réaliser le framework et d'autre part le développeur d'application (*application developer*) qui est la personne qui va personnaliser le framework ([Beck93], [Nier92], [Wirf90]).

3.3.2.1. L'activité de l'ingénieur d'application

Pour développer le framework, l'ingénieur d'application doit avoir une bonne connaissance du domaine pour lequel il veut réaliser le framework.

Son rôle sera d'une part de développer les composants réutilisables et d'autre part de les intégrer dans le framework.

A l'heure actuelle, la conception de frameworks est essentiellement une activité créative et empirique pour laquelle il n'existe pas de support formel, ni en terme de méthodologies de développement ni en terme d'outils pour supporter cette activité ([Beck93]).

Le développement d'un framework est un processus itératif et évolutif, ce qui explique pourquoi un framework peut prendre un certain temps pour se stabiliser ([Beck93], [Nier92], [Wirf90]).

3.3.2.2. L'activité du développeur d'application

Le rôle du développeur d'application est de personnaliser un framework afin d'obtenir une solution particulière. Cette personnalisation se fera, comme toute activité de réutilisation, en quatre étapes ([Bigg87]) qui sont:

(1) la sélection de composants

(2) la compréhension de ces composants

(3) l'affinage de ces composants si nécessaire

(4) l'assemblage des composants afin de définir le comportement général de l'application

Il est clair qu'il est nécessaire d'avoir des outils qui vont aider le développeur d'application dans cette tâche de personnalisation, notamment pour retrouver, affiner et assembler les composants.

A l'heure actuelle, il existe un certain nombre d'outils qui permettent de personnaliser un framework, mais la plupart d'entre eux sont dédiés à un framework bien particulier. Ce qu'il faut réellement, ce sont des outils qui puissent être utilisés pour réaliser des applications à partir de n'importe quel framework ([Beck93]).

Si on reprend notre objectif initial qui est de développer rapidement une application ayant une interface graphique qui soit telle que l'utilisateur la désire, on constate qu'une solution *idéale* serait de faire jouer à un utilisateur le rôle du développeur d'application.

On aurait alors l'utilisateur qui personnaliserait un framework afin d'obtenir une application qui soit réellement comme il la désire.

En ce qui concerne l'aspect rapidité de développement, il serait favorisé par le fait que l'approche framework permet de réutiliser des composants plutôt que de développer une application "à partir de zéro".

Cependant, un utilisateur a, en général, des compétences techniques assez faibles. Donc, pour qu'il puisse jouer le rôle de développeur d'application, il est nécessaire que son activité se limite à spécifier l'application qu'il désire obtenir, et cela, indépendamment de préoccupations d'implémentation.

Pour réaliser cela, le framework doit disposer d'une interface qui va aider le développeur d'application à personnaliser la solution générique.

Nous allons maintenant décrire les trois parties constitutives d'un framework telles qu'elles ont été vues dans [Beck93], nous y détaillerons ce qu'on entend par l'interface d'un framework.

3.3.3. Les trois parties constitutives d'un framework

Comme on vient de le voir, pour que le développeur d'application puisse personnaliser le framework sans se préoccuper des problèmes d'implémentation technique, le framework doit nécessairement lui offrir une interface.

En fait, la solution générique du framework ne sera accessible au développeur d'application que par l'intermédiaire de cette interface.

Cette interface poursuit un double but.

D'une part, elle va fournir une présentation de la solution générique au développeur d'application. Cette présentation étant compatible avec son univers cognitif. A ce niveau, une présentation visuelle est fondamentale pour une bonne compréhension de la solution générique.

D'autre part, l'interface va procurer une guidance au développeur d'application afin de l'aider lors de la personnalisation de la solution générique.

On le voit, l'interface d'un framework correspond en fait à un outil de personnalisation pour ce framework. Cet outil va permettre au développeur d'application de personnaliser le framework à travers un processus graphique interactif ([Beck93]).

Comme le montre la figure 3.1, un framework est composé de trois parties. On a d'une part la sémantique qui correspond à la solution générique pour une classe de problèmes donnés. On a ensuite la présentation et la guidance qui constituent à elles deux, comme on vient de le voir l'interface du framework.

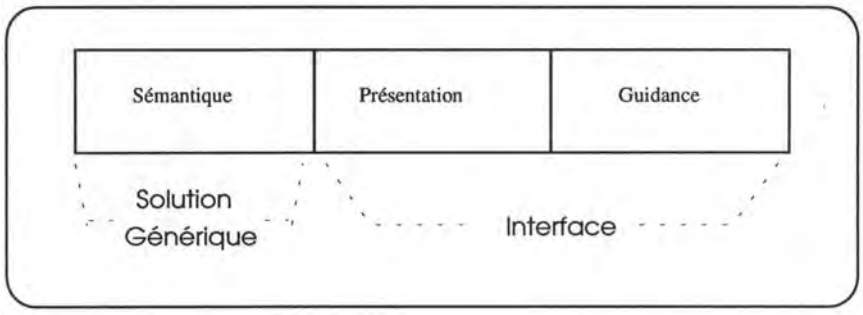


Figure 3.1 : les trois parties d'un framework

3.3.3.1. La sémantique

Le but de la sémantique d'un framework est de fournir, comme on l'a déjà dit, une solution générique pour une classe d'applications, cette solution pouvant être personnalisée par le développeur d'application afin d'obtenir une application particulière.

La solution générique peut en fait être vue comme une application de haut-niveau qu'il faudra affiner.

3.3.3.2. La présentation

Le but de la présentation est d'offrir au développeur d'application une présentation visuelle de la solution générique qui soit compatible avec son univers cognitif, c'est à dire avec laquelle il soit familier.

On veut donc que le développeur d'application ait une compréhension intuitive de la solution générique sans se préoccuper du fait qu'il s'agit, techniquement, d'un ensemble de classes qui interagissent.

3.3.3.3. La guidance

Le but de la guidance est d'aider le développeur d'application lors de la personnalisation du framework.

En fait , cette guidance est fournie sous la forme de scénarios d'interaction qui définissent le dialogue entre le développeur d'application et le framework pour sa personnalisation ([Beck93]).

Pratiquement, la guidance fournit au développeur d'application des directives qui orientent la sélection, l'affinage et l'assemblage des différentes classes du framework lors de sa personnalisation.

Cette guidance peut prendre plusieurs formes qui peuvent aller d'une documentation informelle, jusqu'à une connaissance formelle qui est encodée dans la description du framework ([Beck93]). L'idéal étant d'intégrer totalement cette guidance dans l'interface du framework.

3.4. Conclusion

Nous avons vu que l'on place beaucoup d'espoir dans la réutilisation de composants afin d'améliorer la qualité et la rapidité de développement des logiciels. Un moyen de mettre en oeuvre cette réutilisation peut se faire grâce aux frameworks.

Un framework correspond à une solution générique qui sera personnalisée afin de répondre à un problème particulier. De plus, il est idéal qu'un framework dispose d'une interface permettant de cacher les problèmes techniques liés à sa personnalisation.

Chapitre 4

Chapitre 4 : Présentation de PowerTalk

4.1. Introduction

Depuis ces dix dernières années, on a assisté à l'émergence de ce que l'on appelle les outils CASE (*Computer-Aided Software Engineering*). Il s'agit d'outils informatiques ayant pour but d'apporter un soutien dans les différentes phases du développement de logiciels ([Atte92]).

Traditionnellement, on considère qu'il y a trois catégories d'outils CASE : les *Upper* CASE, les *Lower* CASE et les *Integrated* CASE (I-CASE) ([Atte92], [Grem92]).

Les Upper CASE sont les outils qui apportent un soutien au niveau de la conception d'une application, tandis que les Lower CASE sont les outils qui apportent un soutien au niveau du développement et de la maintenance des applications.

Les Integrated CASE étant quant à eux les outils qui apportent un soutien tout au long du développement d'une application. Il s'agit donc d'outils qui réalisent les fonctionnalités des Upper CASE et des Lower CASE de façon intégrée, c'est à dire que les résultats d'un outil deviennent les données de l'outil suivant, pour finalement aboutir à la production de code ([Atte92]).

Le but poursuivi lors du développement de l'outil PowerTalk était de créer un Integrated CASE, plus précisément de coupler un Upper CASE à un L4G. Comme le montre la figure 4.1, il s'agissait de coupler l'Upper CASE GraphTalk avec le L4G PowerBuilder.

Pour réaliser ce couplage, on va récupérer des informations provenant de la modélisation avec la méthode Merise 2 de GraphTalk. On utilisera ces informations afin d'automatiser la production d'une partie du code d'une application qui sera récupéré dans le L4G PowerBuilder.

En ce qui concerne GraphTalk, il s'agit d'un méta-outil, c'est-à-dire un outil permettant de générer des outils CASE supportant une quelconque méthode. On modélise graphiquement la méthode pour laquelle on veut réaliser un outil CASE, et l'on obtient ainsi un méta-modèle, qui, une fois compilé peut être utilisé.

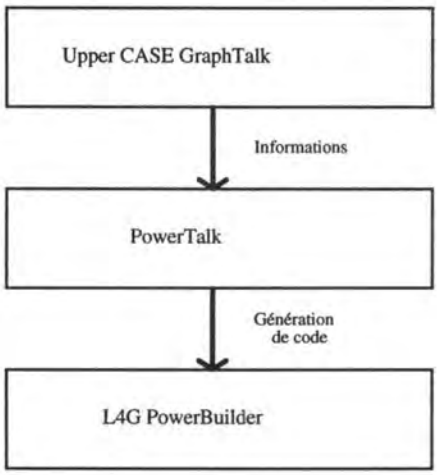


Figure 4.1 : le projet PowerTalk, obtenir un I-CASE

Cependant, il existe déjà divers méta-modèles préexistants supportant différentes méthodes telles que : Merise 2 (Merise Client/Serveur), Merise, Axial, Sadt, Ossad, OMT,...

En ce qui concerne le couplage, il est réalisé à partir de la méthode Merise 2.

Cependant, l'idée originale du projet est de faire en sorte que PowerTalk soit le plus indépendant possible, à la fois de la méthode supportée par GraphTalk, et à la fois par rapport au L4G dans lequel on génère le code. De façon à avoir peu de modifications à faire pour pouvoir supporter d'autres méthodes ou générer dans d'autres L4G.

On veut faire en sorte de créer une architecture qui soit la plus ouverte possible.

Il faut donc considérer qu'à ce niveau, même si certains choix d'implémentation qui ont été faits sont propres à la méthode Merise ou principalement au L4G PowerBuilder, PowerTalk est un produit qui existe indépendamment de GraphTalk et de PowerBuilder.

Nous allons cependant présenter le produit couplé à la méthode Merise, même si, comme nous le verrons, elle est loin d'être adaptée ou du moins bien utilisée pour aider à l'automatisation de la production de code.

Il faut donc toujours bien garder à l'esprit que Merise est une méthode permettant de fournir des informations à PowerTalk, et que cette méthode pourrait (devrait) être toute autre.

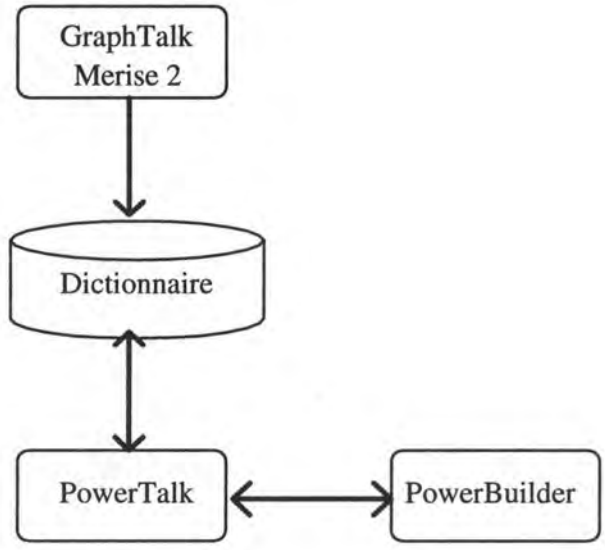


Figure 4.2 : Architecture du couplage

Comme le montre l'architecture du couplage représentée à la figure 4.2, le passage d'informations entre GraphTalk et PowerTalk va se faire grâce à l'utilisation d'un **dictionnaire** intégré à PowerTalk. Les informations qui y seront stockées serviront à l'automatisation de la production de code PowerBuilder. Ce dictionnaire contient également des informations particulières au fonctionnement de PowerTalk.

L'automatisation de la production de code PowerBuilder par PowerTalk va se faire en utilisant le concept de framework. C'est à dire que des ingénieurs d'application qui connaissent bien PowerBuilder vont développer des frameworks

dans l'environnement PowerBuilder et les fournir à PowerTalk qui va les stocker dans son dictionnaire.

Ces frameworks seront ensuite personnalisés par le développeur d'application dans l'environnement de PowerTalk, afin d'obtenir du code PowerBuilder correspondant à une application particulière qui soit telle qu'il la désire. Une partie de cette personnalisation se fera alors automatiquement grâce aux informations fournies par l'Upper CASE GraphTalk.

4.2. Présentation des différents éléments de l'architecture du projet

4.2.1. Introduction

Comme le montre la figure 4.2, l'architecture du projet PowerTalk se compose de quatre éléments. Il y a d'une part l'Upper CASE GraphTalk supportant la méthode Merise 2, il y a l'outil PowerTalk et son dictionnaire intégré et il y a finalement le L4G PowerBuilder.

Nous allons maintenant présenter les outils GraphTalk et PowerBuilder. L'outil PowerTalk et son dictionnaire étant développés séparément aux points 4.3 et 4.4.

4.2.2. GraphTalk

4.2.2.1. Un support aux méthodes

De nombreuses méthodes ont été proposées durant ces vingt dernières années pour améliorer le processus de conception et de développement de logiciels. Depuis une dizaine d'années, de nombreux outils sont apparus sur le marché afin de supporter l'une ou l'autre de ces méthodes. On appelle ces outils des outils CASE ([Hrus91]).

4.2.2.2. Une nouvelle génération d'outils : les méta-outils

La mise en oeuvre d'un outil CASE supportant une méthode spécifique nécessite un investissement considérable en effort et en technologie. Pour

contourner ce problème, une nouvelle génération d'outils est apparue. Il s'agit des **méta-outils** tels que GraphTalk.

Les méta-outils permettent de générer des outils CASE à partir des spécifications formelles d'une méthode et de ses modèles associés. La mise en oeuvre d'un outil CASE à l'aide d'un méta-outil est alors réduite à la déclaration de son formalisme.

On constate donc que grâce aux méta-outils, on développe un outil CASE de façon similaire à son utilisation, c'est à dire en modélisant. Dans ce cas, on dira que l'on méta-modélise.

De plus, grâce à ces outils, l'adaptation d'une méthode supportée par un outil CASE revient simplement à modifier son méta-modèle.

Mais outre le fait que GraphTalk permette de développer des outils CASE, il constitue également un environnement qui va permettre de les utiliser.

A ce niveau, GraphTalk offre une interface graphique conviviale pour l'utilisation des différents modèles de la méthode. Il dispose également d'un dictionnaire qui va contenir les différentes données fournies par l'utilisateur lorsqu'il modélise.

GraphTalk fournit un langage de requête qui permet aux utilisateurs d'interroger ce dictionnaire dans une syntaxe qui est proche de SQL. Il offre également un générateur de documentation.

Finalement, GraphTalk fourni une interface de programmation en C qui permet d'accéder à son environnement de façon programmée, ce qui est très utile pour le couplage avec d'autres outils.

On constate donc que grâce aux méta-outils on peut développer son propre méta-modèle qui va permettre de supporter la méthode que l'on désire.

Cependant, il existe déjà une série de méta-modèles qui supportent différentes méthodes telles que Merise, Merise 2, AXIAL, SADT, OSSAD, OMT, ...

En ce qui concerne PowerTalk, il est couplé avec la méthode Merise 2 (Merise Client/serveur) de GraphTalk.

4.2.2.3. GraphTalk et Merise 2 (Merise Client-Serveur)

4.2.2.3.1. La méthode Merise

La méthode Merise 2 est une adaptation de la méthode Merise afin de tenir compte d'un certain nombre d'évolutions technologiques telles que l'apparition de postes de travail puissants et des applications distribuées.

Nous n'allons pas décrire dans ce mémoire cette méthode pour laquelle il existe peu ou pas de littérature et le lecteur intéressé trouvera une description détaillée de la méthode Merise dans [Gaba89] et [Coll87]. La compréhension de cette méthode étant suffisante pour la compréhension de la suite de ce mémoire puisque le couplage n'utilise pas les apports de la méthode Merise 2.

	Données	*Traitements*
Niveau Conceptuel	**MCD** Modèle Conceptuel des Données	**MCT** Modèle Conceptuel des Traitements
Niveau Logique	**MLD** Modèle Logique des Données	**MOT** Modèle Organisationnel des Traitements
Niveau Physique	**MPD** Modèle Physique des Données	**MOpT** Modèle Opérationnel des Traitements

Figure 4.3 : les modèles de Merise

Comme le montre la figure 4.3, la méthode Merise est une méthode de conception de systèmes d'informations qui propose une démarche qui est menée parallèlement sur les données et les traitements avec en plus une description du système d'information par niveau : niveau conceptuel, niveau logique ou organisationnel et niveau physique ou opérationnel ([Gaba89], [Coll87]).

A chaque niveau correspondent des modèles pour la représentation des données et des traitements. On trouvera une description de ces différents modèles dans [Coll87] et [Gaba89].

Au niveau **conceptuel**, on se préoccupe des finalités de l'entreprise en faisant abstraction des contraintes d'organisation et techniques. Il s'agit essentiellement de décrire le *"Quoi"* ([Gaba89]).

Au niveau **logique**, on prend en compte les choix d'organisation pour lesquels l'analyste se verra attribuer une marge de manoeuvre plus importante. C'est ainsi qu'il précisera les postes de travail, la chronologie des opérations, les choix d'automatisation, tout en intégrant les contraintes éventuelles ([Coll87]). A ce niveau, on décrit le *"Qui fait quoi et où"* ([Gaba89]).

Au niveau **physique**, les choix techniques sont définis. Ce niveau se préoccupe du *"comment faire"* ([Gaba89]).

4.2.2.3.2. Le soutien apporté par GraphTalk

GraphTalk apporte plusieurs soutiens lors de la modélisation. Tout d'abord, il apporte un support graphique pour la création visuelle des modèles. Ensuite, il effectue certaines vérifications de validité des modèles qui sont réalisés par le modélisateur. Il crée aussi automatiquement de la documentation.

Ensuite, le *mapping* entre le MCD et le MLD est effectué automatiquement par GraphTalk. Le MCD correspond au modèle Entité-Association et le MLD correspond au modèle relationnel.

De plus, comme le montre la figure 4.4, GraphTalk permet également de créer automatiquement sur un SGBD la structure de la base de données

correspondant au MLD. Cette création se fait à travers l'utilisation d'ODBC (ODBC est décrit au point 2.2.3).

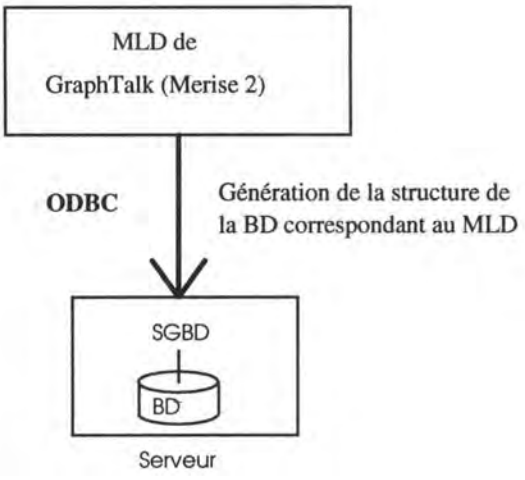


Figure 4.4. : génération de la structure de la BD à partir de GraphTalk

4.2.2.3.3. Adaptation de la méthode pour le couplage

En ce qui concerne le couplage, on va utiliser deux modèles de Merise, et en créer deux nouveaux permettant de faire le lien avec PowerTalk en vue de l'automatisation de la production de code.

Les deux modèles utilisés sont le MLD et le MOT, et les deux nouveaux modèles sont le modèle visible des données (MVD) et le modèle visible des traitements (MVT).

Le MLD correspond au modèle relationnel où l'on retrouve les concepts de relation (appelée plus couramment table) et d'attribut.

Le MVD étant quant à lui le MLD enrichi de la notion de *schéma de relation* qui est composé de divers *constituants*. Un constituant correspondant à un attribut d'une table. On peut alors avoir, au sein d'un même schéma de relation des constituants provenant de différentes tables. Cette notion

correspond donc au concept de vue sur les données, cette vue pouvant porter sur une table ou sur une jointure de tables.

Il est très important de signaler que le **terme** schéma de relation tel qu'il est utilisé dans ce MVD est **incorrect**. En effet, le terme schéma de relation est un terme qui est propre au modèle relationnel et qui correspond à la définition d'une relation, c'est à dire à son expression conceptuelle ([Boda89]). Il nous semble donc qu'un terme comme *vue* eut été plus approprié.

C'est grâce à cette notion de schéma de relation que l'on va pouvoir automatiser la production de code se chargeant de l'accès aux données. Cet accès aux données se faisant, comme nous le verrons plus loin, grâce aux datawindows de PowerBuilder. Concrètement, on aura la création d'une datawindow par schéma de relation.

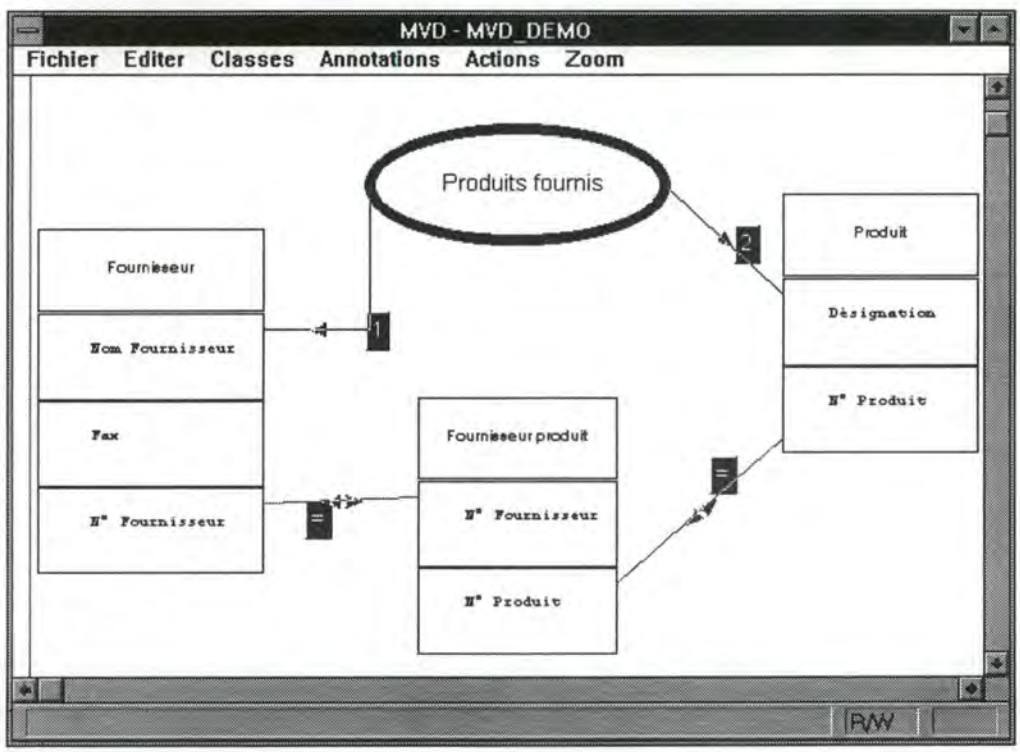


Figure 4.5 : un exemple de schéma de relation dans un MVD

Par exemple, si on a les tables *fournisseur* et *produit* ayant les attributs représentés à la figure 4.5. Et si on sait que dans l'application que l'on veut développer on va avoir besoin de travailler sur la vue permettant d'obtenir la désignation des produits des différents fournisseurs , alors on va créer un schéma de relation *produits fournis* ayant les constituants *nom* du fournisseur et *désignation* du produit. On voit que dans ce cas, il s'agit d'une jointure de tables qui est réalisée par l'intermédiaire de la table *fournisseur produit*.

Au niveau du MOT, on retrouve la notion de procédure fonctionnelle qui est déclenchée par un ou plusieurs événements et qui produit un résultat. Pour la description du MOT, voir [Coll87] et [Gaba89].

On aura alors un MVT par procédure fonctionnelle du MOT. C'est grâce à ce MVT que l'on fera le lien avec PowerTalk. Un MVT donnant lieu à un *module* au niveau de PowerTalk.

Dans ce modèle, on retrouve les notions de fonction prototype, fonction réutilisable et de schéma de relation. Une fonction prototype pouvant utiliser des fonctions réutilisables et des schémas de relation.

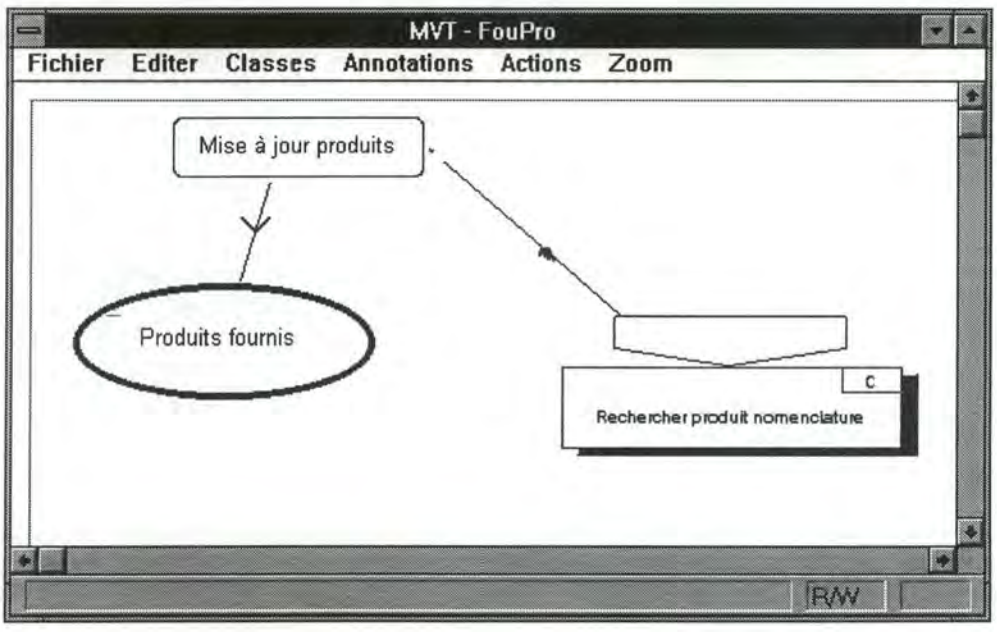


Figure 4.6. : un exemple de MVT

Dans l'exemple de la figure 4.6, on a un MVT contenant une fonction prototype *mise à jour produits* qui utilise le schéma de relation *Produits fournis* et la fonction réutilisable *rechercher produit nomenclature*.

A une fonction prototype correspondra une fenêtre au niveau de PowerBuilder. On trouvera alors dans cette fenêtre un certain nombre de datawindows correspondant aux schémas de relation qu'elle utilise. Cette fenêtre pouvant faire appel à une ou plusieurs autres fenêtres ou fonctions correspondant aux fonctions réutilisables.

Grâce à ces deux modèles, on va obtenir la notion de module qui se retrouvera au niveau de PowerTalk. Un module étant composé de fonctions prototypes, fonctions réutilisables et schémas de relation. On verra comment cela est mis en oeuvre pour automatiser la production de code PowerBuilder au point 4.3.3.2.2.

4.2.3. PowerBuilder

Comme on l'a déjà vu dans le chapitre 2, PowerBuilder est un L4G qui permet de développer des applications ayant une interface graphique en mode client-serveur.

4.2.3.1. Une application est constituée d'objets

En fait, il est primordial de comprendre qu'une application développée à l'aide de PowerBuilder est constituée **uniquement** d'objets qui interagissent et qui sont intégrés dans l'environnement de développement. Ces objets ne sont pas accessibles sous la forme d'un texte comme dans un langage de troisième génération mais sont accessibles par l'intermédiaire de *painters* de manière visuelle. Nous avons déjà montré un exemple d'un tel *painter* au chapitre 2, lorsque nous avons parlé du *window painter* qui permet de créer des fenêtres.

Dans l'environnement PowerBuilder, il existe sept types d'objets différents qui sont:

• les fenêtres : il s'agit de l'interface principale entre l'utilisateur et l'application. Comme nous l'avons vu, ces fenêtres sont accessibles grâce au *window painter.*

• les menus : ces objets sont construits grâce au *menu painter.* Dans ces menus, on va spécifier les items de menu que l'on veut retrouver dans la barre de menu, ainsi que les éventuels items de sous-menu. On trouvera également les actions à accomplir en réaction à la sélection de l'un ou l'autre item.

• les user object : il s'agit d'objets réutilisables que l'on peut construire pour réaliser des traitements généralisés utilisés fréquemment dans les applications. Ces objets sont réalisés grâce au *user object painter.*

• les applications : il s'agit d'objets qui constituent le point d'entrée dans une application. On aura un objet application par application. Cet objet se charge, entre autre, d'initialiser l'activité d'une application. Typiquement, il s'agira de faire appel à la fenêtre principal de l'application. Ces objets sont définis grâce au *application painter.*

• les datawindows : on a vu en détail au point 2.3.6 la description des datawindows. Ces objets sont définis grâce au *datawindow painter.*

• les structures : il s'agit d'un ensemble de une ou plusieurs variables du même type ou de types différents qui sont regroupées sous le même nom et qui peuvent être manipulées comme si elles ne faisaient qu'une. Les structures correspondent aux records de Pascal et de COBOL.

• les fonctions : il est toujours possible à un développeur de créer sa propre fonction. Cela est réalisé grâce au *function painter.*

4.2.3.2. Ouverture de PowerBuilder

A priori, PowerBuilder constitue un environnement de développement qui est fermé par rapport à l'extérieur puisque tous les objets sont accessibles à partir des *painters* de manière visuelle.

On voit que cela représente un problème si l'on veut produire du code PowerBuilder à partir d'un outil autre que PowerBuilder et l'intégrer ensuite dans l'environnement de développement.

Pour faire face à ce problème, l'éditeur de PowerBuilder a développé une interface offrant des fonctions permettant notamment d'exporter et d'importer des objets PowerBuilder de manière programmée. Grâce à cette interface, un programmeur pourra extraire des objets PowerBuilder de l'environnement et les mettre sous la forme de fichiers texte. De même, il pourra importer dans l'environnement des objets qui se trouvent sous la forme de fichiers texte.

On trouvera en annexe 1 un exemple de fenêtre se trouvant sous forme graphique dans l'environnement PowerBuilder et son équivalent sous forme de texte après exportation.

4.3. PowerTalk

4.3.1. Objectif

Comme on l'a déjà vu, PowerTalk est un outil qui va réaliser l'intégration de deux autres outils : l'outil CASE GraphTalk et le L4G PowerBuilder.

L'objectif de PowerTalk est double. Il s'agit de récupérer une partie des informations fournies lors de la modélisation avec GraphTalk afin d'**automatiser** la production de code PowerBuilder tout en mettant en oeuvre un processus de **réutilisation** d'objets PowerBuilder.

Le code produit est alors importé dans l'environnement de développement de PowerBuilder afin de pouvoir être éventuellement amélioré.

L'aspect réutilisation provient de l'utilisation de frameworks qui seront personnalisés afin d'obtenir une application particulière. La personnalisation étant réalisée grâce aux concepts de modèles et de paramètres propres à PowerTalk. C'est également grâce aux modèles et aux paramètres qu'une partie de cette personnalisation pourra se faire automatiquement.

4.3.2. Restriction au niveau de l'accès aux données

Il est clair qu'une grande partie du code d'une application de gestion concerne le traitement de données. On va donc également se préoccuper, outre de l'automatisation de la production de l'interface de l'application, de l'automatisation de l'accès aux données.

On va immédiatement se placer dans une optique restrictive en ce qui concerne cet accès aux données. C'est à dire que tout accès aux données se trouvant sur une base de données se fera par l'intermédiaire des **datawindows** que l'on a décrit au point 2.3.6.

On constate donc que l'automatisation de l'accès aux données se fera à travers l'automatisation de la production de datawindows.

4.3.3. L'approche framework dans PowerTalk

PowerTalk est un outil qui est conçu pour mettre en oeuvre un processus de réutilisation d'objets PowerBuilder en utilisant le concept de framework. Comme on la vu au chapitre 3, un framework représente une solution générique pour une classe de problèmes donnée, généralement appartenant au même domaine. On aura donc un framework par classe ou gamme d'applications.

Avec cette approche, on peut distinguer deux rôles dans le développement d'une application. On aura d'une part l'ingénieur d'application qui sera chargé de réaliser le framework et d'autre part, le développeur d'application qui va réutiliser et personnaliser le framework afin d'obtenir une application particulière.

Le framework est réalisé dans l'environnement de développement de PowerBuilder, après quoi il est repris dans PowerTalk où l'ingénieur lui ajoute une guidance qui aidera à sa personnalisation ultérieure par le développeur d'application. Nous verrons plus loin comment est réalisée cette guidance.

Comme le montre la figure 4.7, PowerTalk va permettre à l'ingénieur d'application de stocker ensuite son framework dans le dictionnaire de PowerTalk où il pourra être réutilisé, toujours grâce à PowerTalk, par différents développeurs d'application.

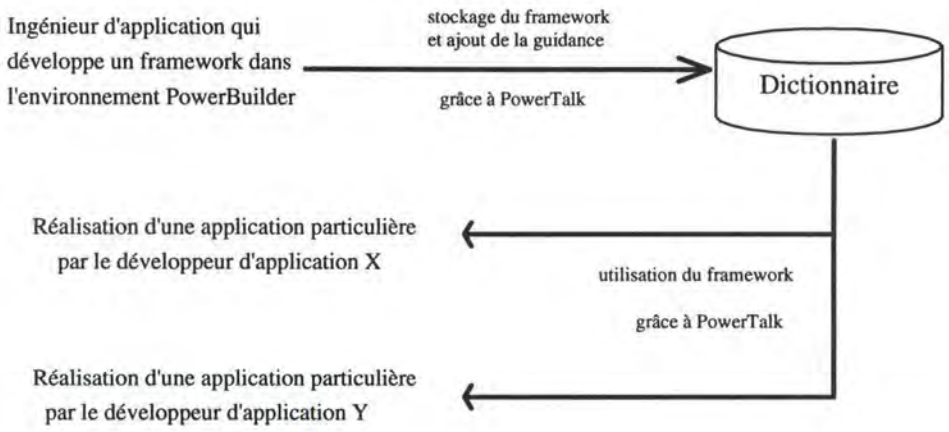


Figure 4.7 : la gestion d'un framework grâce à PowerTalk

Dans un framework utilisé par PowerTalk, on va trouver trois types de composants. Il y a d'une part ce que l'on appelle un squelette d'application, il y a ensuite des objets techniques et il y a finalement des modèles.

1. Le squelette d'application

On aura un squelette d'application par framework. Ce squelette est constitué de certains objets PowerBuilder que l'on retrouvera dans toutes les applications développées avec ce framework. Typiquement, il s'agira d'un objet application, d'un objet fenêtre principale et d'un objet menu principal. On retrouvera déjà au niveau de ce squelette certaines implémentations qui sont communes à toutes les applications que l'on veut développer avec ce framework. Par exemple, l'implémentation dans l'objet menu d'un item de menu d'édition et des sous items permettant de faire du couper-coller.

Ces différents objets interagissent entre eux. Par exemple, l'objet application appelle la fenêtre principale, et cette dernière utilise le menu principal. A ce niveau, on a une application qui fonctionne déjà, mais qui ne fait rien. Il sera nécessaire d'enrichir ce squelette afin d'obtenir une application particulière.

Il est possible pour l'ingénieur d'application, grâce à l'utilisation de PowerTalk, de parametrer ces objets. Les paramètres étant alors remplacés par des valeurs particulières lors de la personnalisation.

2. Les objets techniques

Il s'agit d'objets PowerBuilder qui pourront être réutilisés tels quels dans une application particulière. Ils sont, en principe, inchangés pour toutes les applications qui les utilisent. Un exemple d'un tel objet serait une fenêtre permettant de saisir un mot de passe et un *login*. Ces objets pourront être utilisés par le squelette d'application, mais ils seront également utilisés par les modèles.

3. Les modèles

Nous verrons en détail les modèles au point suivant, mais on peut déjà dire qu'ils correspondent à des objets génériques qui seront personnalisés afin d'être utilisés dans une application particulière. Nous verrons également qu'une partie de cette personnalisation pourra se faire automatiquement grâce à certaines informations récupérées au niveau de GraphTalk.

C'est grâce aux modèles que la production automatique d'objets PowerBuilder est rendue possible.

On verra également que le principe de personnalisation des modèles repose sur le concept de paramètre.

On a vu au chapitre précédent qu'un framework a une guidance dont le but est d'aider le développeur d'application lors de la personnalisation du framework. On peut déjà signaler que cette guidance sera apportée grâce à la notion de paramètre.

4.3.3. Automatisation grâce aux concepts de modèle et de paramètre

Comme on l'a vu au point 4.2.3.2, le seul moyen d'échange entre l'environnement de développement PowerBuilder et l'extérieur va se faire à l'aide d'une interface de programmation permettant d'importer et d'exporter les différents objets sous forme de fichier texte.

On constate donc que le seul moyen d'automatiser la production de code PowerBuilder, c'est-à-dire la production d'objets, sera de travailler sur des fichiers textes que l'on importera par la suite dans l'environnement PowerBuilder.

Pour réaliser cela, on va utiliser des modèles. Un **modèle** correspond au texte d'un objet PowerBuilder dans lequel on trouve des **paramètres**. Lorsque l'on assigne à ces paramètres des valeurs particulières, on obtient alors le texte d'un objet PowerBuilder particularisé en fonction des besoins d'une application. On dira alors que l'on **instancie** des objets PowerBuilder.

Il suffira alors d'importer l'objet instancié vers l'environnement de PowerBuilder pour qu'il puisse s'insérer dans une application particulière.

On peut voir un modèle comme un objet générique qu'il faudra personnaliser, c'est à dire donner de bonnes valeurs aux paramètres, afin qu'il puisse faire partie d'une application particulière.

Avant de donner une description des modèles, nous allons tout d'abord donner une description des paramètres.

4.3.3.1. Description des paramètres

L'utilisation de paramètres dans un modèle est un moyen d'obtenir des objets PowerBuilder qui soient génériques; le remplacement des paramètres par des valeurs particulières permettant alors d'obtenir des objets PowerBuilder qui pourront s'intégrer dans une application.

Les paramètres sont le moyen grâce auquel l'ingénieur d'application pourra doter son framework d'une guidance qui aidera le développeur d'application lorsqu'il réutilisera le framework en vue de produire une application particulière.

Un exemple de paramètre pourrait être la chaîne de caractère contenue dans la barre de titre d'une fenêtre, ou encore le nom que prendra la fenêtre après importation dans PowerBuilder.

4.3.3.1.1. Trois types de paramètres

Les paramètres peuvent être de trois types : automatique, manuel et fonction.

En ce qui concerne les **paramètres automatiques**, il s'agit de paramètres qui vont être remplacés par des valeurs particulières automatiquement par PowerTalk lorsque l'utilisateur demande l'instanciation d'un objet PowerBuilder. Concrètement, il s'agira de remplacer ces paramètres par des valeurs particulières qui se trouvent stockées dans le dictionnaire et qui proviennent de la modélisation avec GraphTalk.

Comme nous le verrons plus loin, le dictionnaire correspond à une base de données qui se trouve sur un SGBD relationnel. On trouvera donc les différentes informations provenant de GraphTalk dans des tables.

Le remplacement d'un paramètre par une valeur particulière consistera à le remplacer par une donnée se trouvant dans une table du dictionnaire.

Le moyen technique utilisé pour arriver à remplacer automatiquement ce paramètre est qu'on lui a associé une requête SQL permettant d'obtenir la bonne valeur dans la table du dictionnaire. Cette requête étant une requête dynamique, c'est à dire une requête contenant des variables dont la valeur sera fournie lors de l'exécution de la requête.

Voici un exemple d'une telle requête :

```
SELECT nom_client FROM client WHERE num_cli = :NUMERO
```

Avec cette requête, on obtiendra le nom des clients pour lesquels le numéro de client correspond au contenu de la variable numéro.

Grâce à ce procédé de requête dynamique, on va pouvoir remplacer un paramètre par une valeur particulière qui dépendra de l'objet particulier que l'on veut instancier.

Les différents paramètres automatiques existants, ainsi que leur requête associée ont été définis une fois pour toute et son disponibles dans PowerTalk. Ces paramètres pourront être utilisés par l'ingénieur d'application afin de réaliser ses modèles.

En ce qui concerne les **paramètres manuels**, il s'agit de paramètres qui seront remplacés par des valeurs particulières manuellement par l'utilisateur de

PowerTalk au moment où il demande l'instanciation d'un objet PowerBuilder. Il s'agira d'informations qui n'ont pu être fournies au niveau de GraphTalk.

Cependant, ces paramètres manuels ont une valeur par défaut qui pourra être utilisée lors de l'instanciation d'objets PowerBuilder. Ils ont également une description qui devra aider le développeur d'application qui est chargé de lui donner une valeur particulière.

La création de ces paramètres manuels, de leur valeur par défaut et de leur description se fera grâce à PowerTalk par l'ingénieur d'application en fonction de ses besoins.

Finalement, les **paramètres de type fonction** sont des paramètres auxquels est associée une fonction. Lors de l'instanciation, si PowerTalk rencontre un de ces paramètres, il exécutera la fonction et remplacera le paramètre par le résultat de l'exécution de la fonction.

Nous avons déjà parlé des datawindows. Il s'agit d'objets qui vont s'insérer dans une fenêtre. On aura donc dans le texte correspondant à cette fenêtre exportée une référence vers l'un de ces objets. Il est également possible de mettre en paramètre cette référence. On appellera ce paramètre un contrôle. Il s'agit d'un paramètre particulier puisqu'il est constitué lui-même d'un texte contenant plusieurs paramètres.

Grâce à ce type de paramètres, on va pouvoir automatiquement lier un objet fenêtre à un objet datawindow particulier que l'on aura instancié au préalable.

4.3.3.1.2. Les paramètres permettent d'apporter la guidance au framework

Puisque les paramètres correspondent aux parties des objets génériques qui devront être adaptées afin d'obtenir un objet particulier, on peut considérer qu'ils constituent la guidance qui va aider le développeur d'application à personnaliser le framework.

Cette guidance prenant même la forme d'une automatisation grâce aux paramètres automatiques et de type fonction.

On a vu que les paramètres manuels disposent d'une description et d'une valeur par défaut. Ce sont ces deux éléments qui vont guider le développeur lorsqu'il devra remplacer ces paramètres par des valeurs propres à l'objet qu'il désire obtenir.

4.3.3.2. Description des modèles

Comme on l'a déjà dit, un modèle est le texte d'un objet PowerBuilder dans lequel on trouve des paramètres. Le remplacement de ces paramètres par des valeurs particulières permet alors d'obtenir le texte d'un objet PowerBuilder particularisé en fonction des besoins d'une application. On dira alors que l'on instancie des objets PowerBuilder.

Il est possible de réaliser des modèles pour les sept types d'objets PowerBuilder que l'on a vu au point 4.3.3.1. Mais le plus souvent, on réalisera des modèles de fenêtres et de datawindows.

On trouvera, en annexe 2, un exemple de texte constituant un objet datawindow de PowerBuilder et un exemple de texte constituant un modèle pour une datawindow. Comme on vient de le voir, un modèle est le texte d'un objet PowerBuilder contenant des paramètres. Pour reconnaître les paramètres du reste du texte, on a choisi de les placer entre les symboles "<<" et ">>".

Avant de voir comment on instancie les modèles afin d'automatiser la production d'objets PowerBuilder, nous allons voir comment réaliser ces modèles.

4.3.3.2.1. Réalisation des modèles

La création de modèles est une opération qui sera réalisée par l'ingénieur d'application. Il s'agit d'une personne qui connaît PowerBuilder et qui a une bonne connaissance du domaine d'application pour lequel il veut réaliser un modèle (c'est à dire un objet générique).

Pour créer le modèle d'un objet de PowerBuilder, on va tout d'abord créer un objet dans l'environnement de PowerBuilder.

On va créer cet objet de la même manière que si on désirait l'utiliser dans une application spécifique en prenant soin cependant qu'il soit réalisé de manière à pouvoir être réutilisé dans d'autres applications sans avoir trop de changements à lui apporter.

Il faut donc développer un objet en le pensant comme un objet générique mais tout en le particularisant déjà.

Lorsque cet objet est développé, il est exporté en dehors de l'environnement de PowerBuilder sous la forme d'un fichier texte. Ce texte est alors repris dans l'environnement de PowerTalk par l'ingénieur d'application qui peut alors le transformer en modèle.

Pour réaliser ce modèle, il lui faut remplacer dans le texte de l'objet les parties qu'il considère comme variables par des paramètres. Comme le montre la figure 4.6, PowerTalk lui apporte une aide à ce niveau en lui offrant la possibilité de coller dans le texte les paramètres qu'il désire en choisissant parmi la liste des paramètres existants.

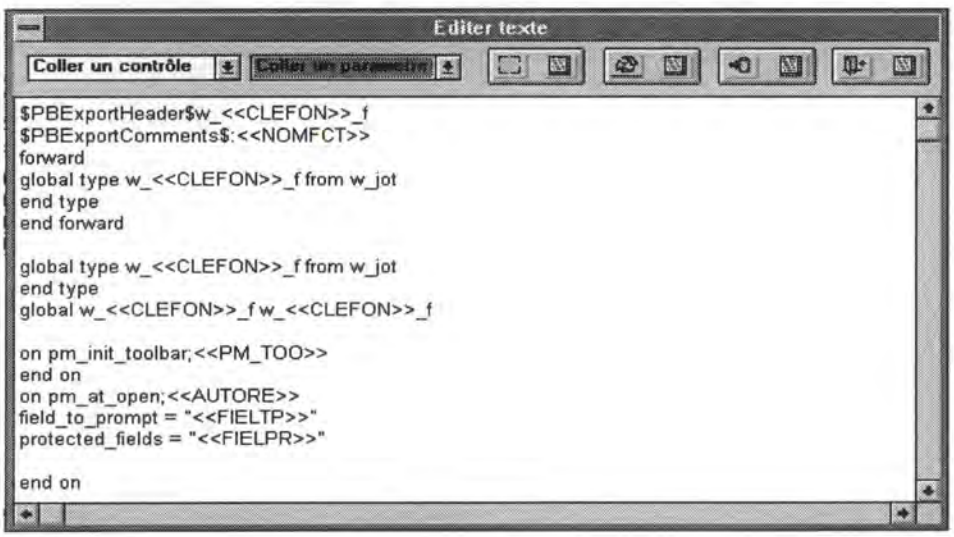


Figure 4.6: la création d'un modèle avec PowerTalk

Lorsque le modèle est créé, il est alors stocké dans le dictionnaire de PowerTalk.

4.3.3.2.2. Instanciation des modèles

Comme il est illustré à la figure 4.7, un modèle est conçu de manière à pouvoir être réutilisé afin d'instancier différents objets PowerBuilder selon les valeurs particulières que prendront ses paramètres.

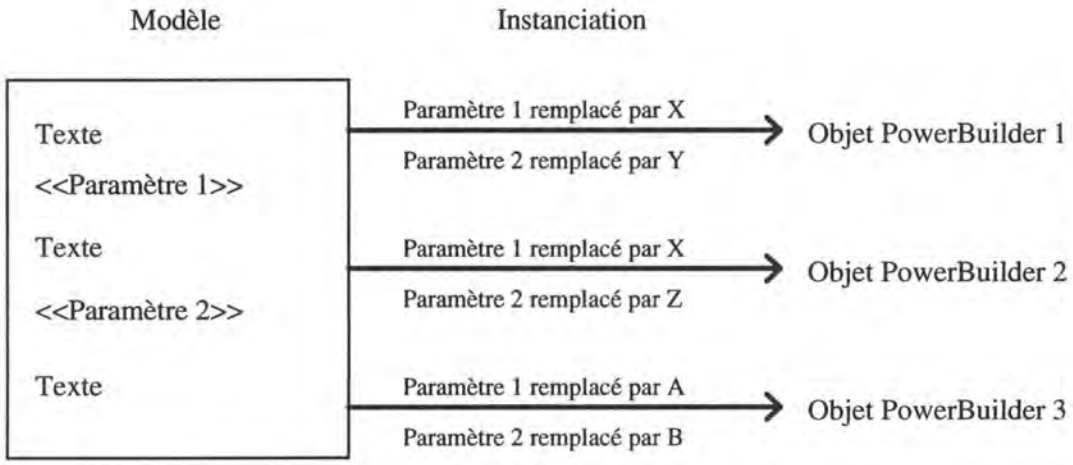


Figure 4.7 : un modèle permet d'instancier plusieurs objets

L'utilisation de modèles en vue de l'instanciation d'objets PowerBuilder va se faire grâce à l'utilisation de PowerTalk.

On a vu que l'utilisation de Merise permet d'obtenir la notion de schéma de relation et de module contenant des composants qui peuvent être des fonctions prototypes, des fonctions réutilisables et des schémas de relation. On a également vu qu'on a un module par procédure fonctionnelle du MOT. Les différents modules qui constituent le MOT et les différents schémas de relation sont descendus au niveau de PowerTalk par l'intermédiaire du dictionnaire.

Ces grâce aux informations que contiennent les modules et les schémas de relation que va se faire l'instanciation des modèles.

Pour réaliser l'instanciation, le développeur d'application va associer un modèle à chaque composant du module. Typiquement, on associera un modèle de fenêtre à une fonction prototype et un modèle de datawindow à un schéma de relation. Pour les fonctions réutilisables, on pourra associer un modèle de fenêtre ou un modèle de fonction.

Pour chaque modèle associé à un composant, PowerTalk présente au développeur d'application la liste des paramètres manuels qu'il contient, la description de ces paramètres ainsi que leur valeur par défaut. Il doit alors fournir pour chaque paramètre une valeur particulière.

Comme le montre la figure 4.8, lorsque l'utilisateur demande l'instanciation d'un modèle, PowerTalk va remplacer les paramètres du modèle par des valeurs particulières.

Les paramètres manuels sont remplacés par les valeurs particulières fournies au préalable par l'utilisateur.

Les paramètres automatiques sont, quant à eux, remplacés par des valeurs particulières par PowerTalk grâce aux informations qui se trouvent dans le dictionnaire et qui sont propres au composant auquel le modèle est associé.

Finalement, les paramètres de type fonction sont remplacés par PowerTalk par le résultat de l'exécution de la fonction à laquelle ils correspondent.

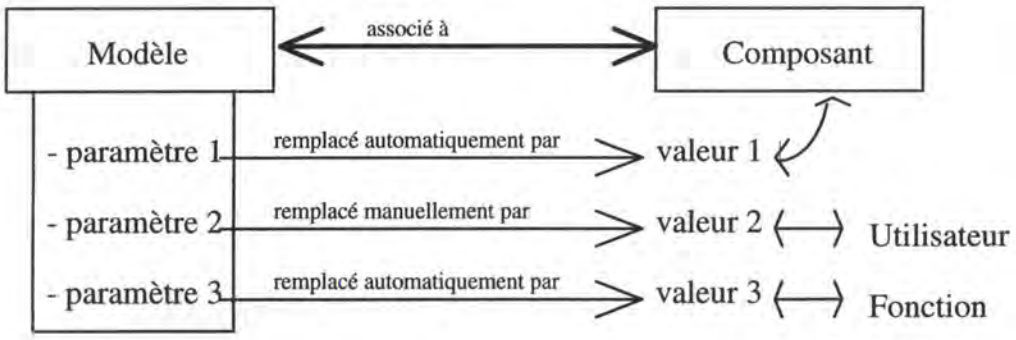


Figure 4.8 : processus d'instanciation d'un modèle

On a également vu qu'au sein d'un module, les composants interagissent. On a par exemple une fonction prototype qui utilise un ou plusieurs schémas de relation.

Au niveau de PowerTalk, cela va donner lieu à une fenêtre qui utilise une ou plusieurs datawindows. Le lien entre une fonction et le ou les schémas de relation qu'elle utilise sera réalisé grâce aux paramètres spéciaux de type contrôle que l'on a décrit au point 4.3.3.1.1.

Concrètement, dans le modèle de la fenêtre utilisé pour instancier la fonction prototype, on aura autant de paramètre de type contrôle qu'il n'y a de datawindows utilisées dans cette fenêtre. Avant d'instancier le modèle de la fenêtre, l'utilisateur devra signaler à quel schéma de relation correspond chaque contrôle.

Lors de l'instanciation, PowerTalk se chargera de remplacer les paramètres de type contrôle avec des références correctes aux datawindows qui correspondent aux schémas de relation particuliers auxquels on a lié les contrôles.

4.4. Le Dictionnaire de PowerTalk

Nous avons déjà parlé du dictionnaire de PowerTalk, nous allons maintenant voir ce qu'il contient comme données.

C'est grâce à ce dictionnaire que va se faire le lien entre GraphTalk et PowerTalk. C'est à dire que des données provenant de la modélisation avec GraphTalk vont être stockées dans ce dictionnaire. Elles seront ensuite réutilisées par PowerTalk afin d'instancier automatiquement les modèles.

Une fois ces modèles instanciés, ils seront importés par PowerTalk dans l'environnement PowerBuilder où ils pourront s'intégrer dans une application particulière.

Le dictionnaire contient également les différents frameworks réalisés par les ingénieurs d'application.

Ce dictionnaire est en fait une base de données qui se trouve sur un serveur de données relationnel et PowerTalk et GraphTalk y accèdent grâce à ODBC (voir le

point 2.2.3 pour la description d'ODBC). Ce qui veut dire que ce dictionnaire pourra se trouver sur n'importe quel SGBD qui a développé un driver ODBC (Oracle, SQLServer de Sybase, Ingres, Informix, ...).

4.4.1. Les données provenant de GraphTalk

Les données fournies par GraphTalk et stockées dans le dictionnaire proviennent des deux modèles dont on a parlé au point 4.2.2.3.3, à savoir le modèle visible des données et le modèle visible des traitements. Ces modèles ont été ajoutés au méta-modèle de Merise 2 en vue de faire le lien entre la méthode et l'outil de développement PowerBuilder.

4.4.1.1. Les données provenant du modèle visible des données

Dans ce modèle, on retrouve les concepts de schéma de relation et de constituant qui correspondent à une vue de la base de donnée que l'on va manipuler dans une application particulière (voir le point 4.2.2.3.3).

C'est grâce à ces concepts que l'on va pouvoir automatiser la production de datawindows, c'est à dire automatiser l'accès aux données. En fait, on va avoir la création d'une datawindow par schéma de relation.

Au niveau du dictionnaire, on va récupérer en provenance de GraphTalk les informations sur les schémas de relation ainsi que leurs constituants associés. On connaîtra ainsi le nom d'un schéma de relation, le nom des différentes tables et colonnes qui constituent un schéma, le format d'un constituant (caractère, entier, ...), l'ordre des constituants (pour l'affichage),... Toutes ces informations ayant été fournies au niveau de la modélisation avec GraphTalk et descendues dans le dictionnaire afin de permettre d'automatiser la production de la datawindow.

4.4.1.2. Les données provenant du modèle visible des traitements

Au niveau du MVT, on a vu que l'on avait la notion de module contenant des fonctions prototypes, des fonctions réutilisables et des schémas de relation. Les fonctions prototypes utilisent des fonctions réutilisables et des schémas de

relation. On a également vu que l'on a un module par procédure fonctionnelle du MOT.

Ces notions seront descendues au niveau de PowerTalk où elles serviront à l'instanciation de modèles comme on la vu au point 4.3.3.2.2.

On obtiendra essentiellement comme informations sur un module : son nom, ainsi que le nom des différents composants qu'il contient.

4.4.2. Les frameworks provenant de PowerBuilder

Dans le dictionnaire, on va retrouver les différents frameworks qui ont été développés grâce à l'utilisation combinée de PowerBuilder et PowerTalk.

L'utilisation de PowerTalk permet de fournir la guidance du framework.

Ces frameworks seront à la disposition des différents développeurs d'application qui voudront les utiliser pour réaliser une application particulière.

4.5. Evaluation de PowerTalk

PowerTalk est un produit qui n'est pas encore entièrement opérationnel et qui peut encore évoluer. Il n'existe d'ailleurs à l'heure actuelle aucune documentation et il n'y a aucune aide en ligne fournie à l'utilisateur. Nous allons en faire une évaluation dans l'état où il se trouve actuellement, c'est à dire tel qu'il a été décrit.

4.5.1. Des modèles réutilisables

Comme on l'a vu, PowerTalk permet d'automatiser la production de code PowerBuilder tout en mettant en oeuvre un processus de réutilisation.

PowerTalk n'est pas un outil qui est conçu pour générer du code uniquement à partir de spécifications conceptuelles que l'on fournirais grâce à une méthode. Il s'agit d'un outil qui va permettre d'instancier *presque* automatiquement (puisqu'on a vu que certains paramètres sont manuels) des modèles d'objets existants. La mise en oeuvre de cette automatisation à travers les concepts de paramètres et de modèles ne trouve d'ailleurs sont intérêt que parce qu'elle permet la réutilisation.

Il est clair qu'il n'est pas avantageux de développer un modèle en sachant qu'il sera utilisé pour instancier un seul objet PowerBuilder, puisque la création d'un modèle nécessite au préalable la création d'un objet PowerBuilder (voir le point 4.3.3.2.1). Il est donc vital que l'objet PowerBuilder sur base duquel on va développer un modèle soit suffisamment générique pour pouvoir être réutilisé dans différentes applications ou au sein d'une même application.

Ce sera le rôle de l'ingénieur d'application qui va développer les modèles de veiller à ce que ses modèles soient suffisamment génériques.

4.5.2. Pourquoi automatiser la production d'objets PowerBuilder.

L'utilisation de GraphTalk fournit un soutien dans la phase de conception d'une application, dont le résultat principal est la production de documents qui serviront aux développeurs lors de la phase d'implémentation de l'application.

Lorsque l'on utilise PowerTalk conjointement à GraphTalk, des informations qui ont été fournies lors de l'utilisation de GraphTalk sont utilisées afin d'automatiser une partie de la phase d'implémentation dans PowerBuilder. Cette automatisation doit permettre d'obtenir plus **rapidement** des applications **fiables**, tout en maintenant la **cohérence** entre la phase de conception et celle d'implémentation.

4.5.2.1. Plus grande rapidité de développement

A priori, lorsque l'on considère l'automatisation du développement d'une application, il paraît évident que cela va permettre de développer plus rapidement cette application.

Cependant, les gains de temps ne sont pas si évidents que cela, et il faut considérer les différents facteurs qui entrent en jeu.

D'une part, pour que cette automatisation fonctionne, il faut concevoir des modèles. Et la conception de ces modèles peut demander du temps. Il faudra donc que le temps que l'on gagne à automatiser la production soit supérieur au temps de développement des modèles. A ce niveau, il est primordial que les modèles soient conçus de façon à permettre l'automatisation de la production de

plusieurs objets PowerBuilder particuliers comme on l'a montré à la figure 4.7. C'est à dire qu'ils doivent être conçus de façon à pouvoir être réutilisés.

Ensuite, on a vu que l'automatisation est rendue possible grâce aux informations qui ont été fournies au niveau de la méthode Merise supportée par GraphTalk.

Il faut, là encore, prendre en considération la surcharge d'informations qui doivent être fournies lors de la modélisation et qui sont propres à l'automatisation (comme par exemple le format d'affichage d'un constituant de schéma de relation).

Il n'est pas intéressant de diminuer le temps nécessaire au développement d'une application si pour cela on doit accroître fortement le temps nécessaire à la modélisation.

A ce niveau, il est vital que la méthode supportée par GraphTalk soit adaptée au développement des applications interactives, ce qui n'est malheureusement pas le cas des méthodes traditionnelles [Boda94]. Cette méthode doit également être adaptée au développement d'applications sur base de composants réutilisables.

De plus, à l'heure actuelle, le nombre d'informations qui peut être récupéré de la méthode Merise est relativement faible, ce qui fait que la partie paramétrée des modèles est relativement réduite (en ce qui concerne les paramètres automatiques), et donc la partie produite automatiquement aussi.

Finalement, on a vu que les L4G permettent de développer très rapidement une application. Certains auteurs prétendent qu'un L4G peut augmenter jusqu'à dix fois la vitesse de développement d'une application par rapport à un langage de troisième génération ([Buck90], [Keye92]).

Il est donc clair que lorsque l'on choisit d'automatiser la production d'une application dans un L4G, on doit s'attendre à ce que les gains obtenus au niveau de la rapidité du développement soit moindre que dans un langage de troisième génération.

En conclusion, nous dirons qu'il n'est pas sûr que l'automatisation de la production telle qu'elle est réalisée actuellement grâce à PowerTalk apporte un réel gain au niveau de la rapidité de développement. Et cela, principalement à

cause d'une inadéquation de la méthode grâce à laquelle on obtient des informations servant à l'automatisation du développement de l'application.

4.5.2.2. Plus grande fiabilité des applications

Ici encore, il n'est pas du tout sûr que des objets produits automatiquement grâce à PowerTalk soient fiables. En effet, puisque l'automatisation est rendue possible grâce aux modèles et aux paramètres, la fiabilité des objets produits dépendra de la correcte réalisation des modèles et de la bonne utilisation des paramètres. Ce sera le rôle de l'ingénieur d'application de s'assurer que les modèles soient fiables.

Cependant, puisque les modèles sont faits pour être réutilisés, on peut espérer qu'ils seront corrigés si on constate que les objets qu'ils permettent d'instancier sont incorrects après qu'ils aient été utilisés dans une application particulière.

4.5.2.3. Maintien de la cohérence entre la conception et l'implémentation

Puisque l'automatisation du développement se fait sur base d'informations provenant d'une méthode permettant de concevoir des applications, on peut ainsi obtenir une cohérence entre la phase de conception et la phase d'implémentation.

Si on considère l'utilisation de la méthode Merise de GraphTalk sans l'utilisation de PowerTalk, on aboutit à la production de documents qui seront utilisés par les développeurs afin d'implémenter l'application. Les développeurs peuvent alors prendre certaines libertés par rapport à la solution conceptuelle.

Le fait d'automatiser la production de code sur base des informations conceptuelles leur laisse moins de libertés, ce qui permet d'assurer une meilleure cohérence entre la phase de conception et la phase d'implémentation.

Mais là encore, cela dépendra essentiellement des informations que l'on a pu obtenir de la méthode.

4.5.3. Inadéquation de la méthode Merise

On peut signaler qu'il n'y a eu aucune recherche effectuées par l'éditeur de PowerTalk afin de voir quelle serait la méthode la plus adaptée pour permettre l'automatisation de la production de code dans un L4G. Si on a retenu la méthode Merise 2 et le L4G PowerBuilder, c'est simplement parce que le client qui a demandé l'initialisation du projet PowerTalk disposait de ces deux outils et voulait les intégrer, sans pour autant avoir réfléchi sur la faisabilité du projet.

Il est clair que Merise n'est pas adapté pour le couplage avec un L4G tel qu'il est effectué par PowerTalk, c'est à dire en utilisant des frameworks. La preuve est que l'on a dû rajouter deux modèles uniquement utilisés pour permettre l'automatisation.

De plus, les informations que l'on peut récupérer de ces deux modèles sont très faibles, si on exclu les informations permettant d'automatiser la production des datawindows.

Il faut également signaler qu'il n'y a pas eu de véritable réflexion méthodologique par l'éditeur de PowerTalk sur la méthode Merise 2 qui offre quant même de nouveaux modèles par rapport à Merise traditionnel.

Cependant, comme nous le verrons au point 4.5.8, PowerTalk est relativement indépendant de la méthode et si on dispose d'une méthode plus adaptée au développement d'applications interactives sur base de composants réutilisables, il est tout à fait possible de la coupler très facilement à PowerTalk.

4.5.5. PowerTalk et le reverse engineering

PowerTalk est un outil qui permet d'automatiser la production d'applications sur base d'informations venant d'un outil CASE. Mais que se passe-t-il au niveau de l'outil CASE lorsque une application qui a été produite est modifiée dans l'environnement de PowerBuilder ? C'est ce qu'on appelle le reverse engineering.

Comme le montre la figure 4.8, l'idéal lorsque l'on génère le code d'une application à partir de spécifications conceptuelles, c'est de disposer d'un module qui soit capable, lorsque l'application est modifiée, de répercuter ces modifications au niveau conceptuel.

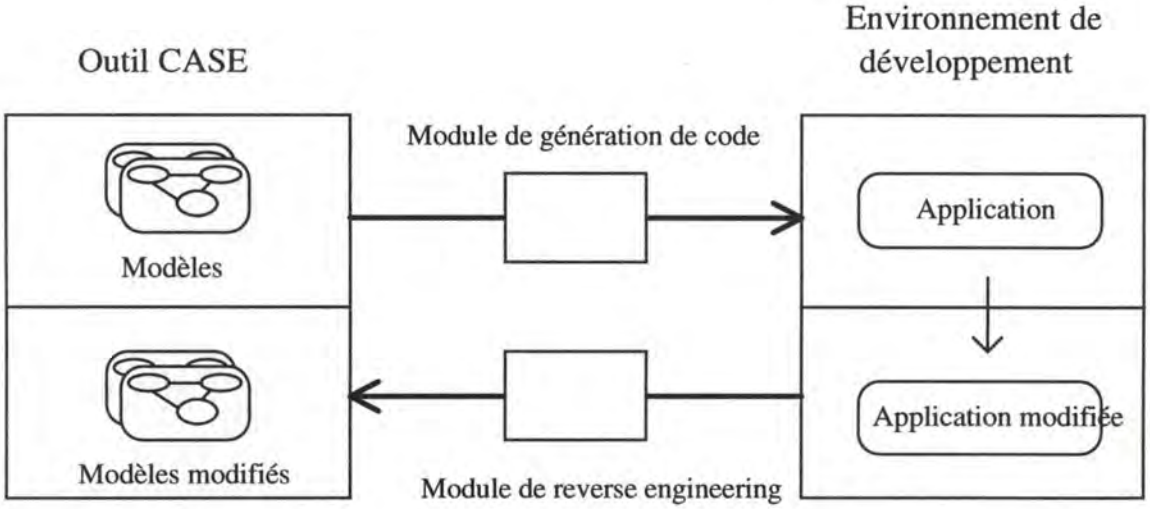


Figure 4.8. : une solution idéale avec un module de reverse engineering

Au niveau de PowerTalk, il n'existe aucun module de reverse engineering. Cela signifie que lorsqu'une application générée est modifiée grâce à PowerBuilder, il n'est pas possible de répercuter les modifications au niveau de l'outil CASE GraphTalk.

Cela pose évidemment un gros problème lorsque l'on désire modifier les modèles correspondant à une application qui a déjà été générée et enrichie. On ne verra aucune trace des enrichissements qui ont été apportés et lorsque l'on générera à nouveau, on obtiendra une application où il faudra à nouveau apporter les améliorations qui avaient déjà été effectuées.

4.5.6. L'accès aux données grâce aux datawindows

On a vu que l'accès aux données dans les applications produites grâce à PowerBuilder est réalisé uniquement à travers l'utilisation de datawindows.

A priori, cela peut paraître très restrictif, mais lorsque l'on connaît la puissance de ces datawindows, on comprend vite ce que l'on peut gagner à les utiliser. En effet, PowerBuilder offre de nombreuses fonctions permettant de manipuler très facilement ces datawindows.

De plus, l'utilisation de datawindows est très intéressante lorsque l'on se place dans une optique de réutilisation.

On a vu au point 2.3.6 que les fonctions s'appliquent sur le datawindow *control* qui se trouve dans une fenêtre, indépendamment de l'objet datawindow. Dès lors, si on travaille sur un objet datawindow bien particulier à travers un datawindow *control* et aux fonctions que l'on peut appliquer sur lui, il suffit de lier ce *control* à un autre objet datawindow pour que toutes les fonctions qui ont été implémentées sur ce *control* restent valables pour le nouvel objet datawindow.

4.5.7. PowerTalk et l'approche framework

Grâce à l'approche framework dans PowerTalk, on peut mettre en oeuvre un processus de réutilisation qui apporte plusieurs avantages.

D'une part, cela permet d'augmenter la fiabilité de l'application produite puisque l'on réutilise des composants qui ont été testés à fond, et plus précisément qui ont fonctionné sur des systèmes opérationnels.

Ensuite, cela permet d'utiliser efficacement des spécialistes à qui on demande de développer les frameworks qui encapsulent leur connaissance, plutôt que de leur faire faire la même chose sur différents projets.

Cette approche permet également d'implémenter des standards que l'on va retrouver dans les frameworks.

Finalement, cela permet de réduire le temps de développement du logiciel puisque la réutilisation de composants permet d'accélérer la production du logiciel et de réduire le temps de validation.

Il est clair que l'utilisation de framework n'est pas particulière à PowerTalk, et peut évidemment se faire uniquement dans l'environnement PowerBuilder. On a d'ailleurs vu que les frameworks utilisés par PowerTalk ont dû être développés au préalable grâce à PowerBuilder.

Cependant, l'utilisation de framework grâce à PowerTalk offre deux avantages par rapport à l'utilisation uniquement avec PowerBuilder.

D'une part, PowerTalk va stocker les différents frameworks dans un dictionnaire qui se trouve sur un serveur de données. Ces frameworks seront alors disponibles pour les différents utilisateurs de PowerTalk qui pourront alors le ramener dans leur environnement de développement. PowerBuilder n'offre pas un tel dictionnaire.

Ensuite, et c'est ce qui est le plus important, c'est que PowerTalk permet, grâce à ses paramètres, de doter le framework d'une guidance qui aidera le développeur d'application à personnaliser le framework.

Nous avons vu au chapitre 3 qu'un framework est composé de trois parties : la sémantique, la présentation et la guidance. Nous allons maintenant voir comment se retrouve ces différentes parties dans les frameworks utilisés par PowerTalk.

4.5.7.1. La sémantique

La sémantique d'un framework correspond à la solution générique qui doit être personnalisée en vue d'obtenir une application particulière.

Il s'agit donc des différents objets PowerBuilder réutilisables qui constituent le framework.

4.5.7.2. La présentation

Le but de la présentation est d'offrir au développeur d'application une présentation visuelle de la solution générique avec laquelle il soit familier.

PowerTalk ne propose pas une telle présentation. Le développeur d'application doit avoir une connaissance des différents objets qui constituent le framework en dehors de PowerTalk. Ainsi, il est nécessaire qu'il connaisse les modèles afin de savoir quel modèle il doit utiliser lorsqu'il désire instancier un objet PowerBuilder qui s'intégrera dans son application.

Cette connaissance ne peut lui être fournie que par l'ingénieur d'application qui a développé le framework. Ce sera donc à l'ingénieur d'application de donner une présentation de son framework, par exemple, sous la forme d'un manuel.

4.5.7.3. La guidance

Le but de la guidance est d'aider le développeur d'application lors de la personnalisation du framework.

On a vu au point 4.3.3.1.2. que la guidance du framework est apportée grâce aux paramètres.

4.5.8. Ouverture de PowerTalk

Comme on l'a déjà dit au point 4.1, l'idée sous-jacente à la création de PowerTalk est de permettre d'avoir une architecture qui soit la plus ouverte possible. C'est à dire que le produit ne soit dépendant, ni de la méthode supportée par GraphTalk, ni du L4G pour lequel on veut automatiser la production de code.

Il faut reconnaître que certains choix d'implémentation ont été faits en fonction de la méthode Merise et surtout du L4G PowerBuilder. Notamment, en ce qui concerne l'utilisation des datawindows.

Cependant, en ce qui concerne GraphTalk, l'ouverture vers les différentes méthodes existantes est belle et bien présente. En fait, pour que PowerTalk puisse s'adapter à une nouvelle méthode, il suffit d'adapter le dictionnaire et de définir de nouveaux paramètres automatiques qui sont fonctions des informations que l'on pourra récupérer dans la méthode.

On voit donc que si on dispose d'une méthode qui soit adaptée à la conception d'applications interactives et qui supporte l'approche framework, on peut espérer que l'utilisation de PowerTalk permette d'automatiser au mieux la production d'application.

Conclusion

Conclusion

L'objectif de ce mémoire était de présenter PowerTalk qui est un outil permettant d'automatiser le développement d'applications ayant une interface graphique. Les applications produites pouvant alors être récupérées dans l'environnement de développement PowerBuilder qui constitue, comme nous l'avons vu, un langage de quatrième génération.

Nous avons vu que dans cet environnement, la programmation est visuelle et événementielle. De plus, il permet de développer des applications qui utilisent des données se trouvant sur un serveur de donnée et il se prête bien au prototypage.

L'automatisation du développement grâce à PowerTalk se fait en utilisant le concept de framework. Un framework représente une application générique pour une classe de problèmes donnée, généralement appartenant au même domaine; cette application pouvant être personnalisée pour répondre à un problème particulier.

La personnalisation d'un framework existant afin d'obtenir une application particulière, sera réalisée à travers à l'utilisation de PowerTalk.

Une partie de cette personnalisation sera alors réalisée automatiquement par PowerTalk. Afin de permettre cette automatisation, il va utiliser des informations qui ont été fournies suite à l'utilisation d'un outil CASE supportant une méthode de conception d'application. Actuellement, ces informations proviennent de la méthode Merise.

Cependant, PowerTalk est un outil qui se veut indépendant de la méthode supportée par l'outil CASE. Il est construit de façon à s'adapter à n'importe quelle méthode.

Mais nous avons vu que toutes les méthodes, et à commencer par les méthode Merise, ne sont pas adaptées pour permettre d'automatiser le développement d'applications en utilisant le concept de framework.

Pour pouvoir utiliser efficacement PowerTalk, il est vital que les informations qu'il utilise afin d'automatiser la personnalisation d'un framework proviennent d'une méthode qui soit adaptée au développement d'applications interactives sur base de frameworks réutilisables.

Bibliographie

Bibliographie

[Atte92] ATTENBOROUGH R., Computer-Aided Software Engineering (CASE) : An Overview, Datapro (Corporate Software & Solutions), 4101, Novembre 1992.

[Beck93] BECKER K., Reusable Frameworks for Decision Support Systems Development, Thesis submitted in fulfilment of the requirements for the degree of doctor of Science (Computer Science Option), Septembre 1993.

[Bers92] BERSON A., Client/Server Architecture, McGraw-Hill, 1992.

[Boda89] BODART F., PIGNEUR Y., Conception Assistée des Systèmes d'Information, Méthode-Modèles-Outils, 2e édition, Masson, 1989.

[Boda94] BODART F., HENNEBERT A-M., LEHEUREUX J-M., PROVOT I., VANDERDONCKT J., ZUCCHINETTI G., Dimensions clé pour une méthodologie de développement d'applications interactives, FUNDP, Projet Trident, 1994.

[Boeh88] BOEHM B., A Spiral Model of Software Development and Enhancement, Computer, May 1988.

[Buck90] BUCKLAND J.A., End-User Languages and Tools, Journal of Critical Issues in Information Processing Management and Technology, Vol. 7, #2, 1990.

[Bigg87] BIGGERSTAFF T., RICHTER C., Reusability Framework, Assessment, and Directions, IEEE Software, Vol 4, #2, Mars 1987.

[Coll87] COLLONGUES A., HUGUES J., LAROCHE B., Merise : Méthode de Conception, Bordas, 1987.

[Gaba89] GABAY J., Apprendre et pratiquer Merise, Masson, 1989.

[Grem92] GREMILLION L., Developing a Repository and CASE Strategy, Datapro (Management of Applications Software), AS07-700, Janvier 1992.

[Gutt93] GUTTMAN M., MATTHEWS J., Client/Server Computing : Emerging Trends, Solutions, and Strategies, Datapro (Management of Applications Software), AS85-400, Janvier 1993.

[Horo84] HOROWITZ E. & MUNSON J., An expensive view of reusable software, IEEE Transactions on Software Engineering, Vol SE-10, #5, Septembre 84.

[Hrus91] HRUSCHKA P., CASE Support for the Software Process, dans Lecture Notes in Computer Science 550, ESEC'91, édité par Van Lamsweerde A. & Fugetta A., Springer-Verlag, 1991.

[Keye92] KEYES J., Management Perspectives on 4GLs as Application Development Toolsets, Datapro (Development Tools & Environments) 2008, Novembre 1992.

[Lee90] LEE E., User-Interface development Tools, IEEE Software, Mai 1990.

[Lefe93] LEFEBVRE A., L'Architecture Client-Serveur : aspects techniques & enjeux stratégiques, Armand Colin, 1993.

[Lint94] LINTHICUM D., 4GL's : Productivity at What Cost ?, DBMS, Vol 7, # 5, Mai 1994.

[Mart85] MARTIN J., Fourth-Generation Languages, Volume I Principles, Prentice-Hall, 1985.

[Mart86] MARTIN J., Fourth-Generation Languages, Volume III 4GLs from IBM, Prentice-Hall, 1986.

[Mein91] MEINADIER J.P., L'interface utilisateur : pour une informatique plus conviviale, Dunod, 1991.

[Micr92] MICROSOFT, Open Database Connectivity, Octobre 1992.

[Nier92] NIERSTRASZ O., GIBBS S., TSICHRITZIS D., Component-Oriented Software Development, Communications of the ACM, Vol 35, #9, Septembre 1992.

[Sacr91] SACRE B., SACRE-PROVOT I., Proposition d'un langage de spécification de l'interface homme-machine d'une application de gestion hautement interactive, FUNDP, rapport interne, Décembre 1991.

[Sacr92] SACRE B., SACRE-PROVOT I., VANDERDONCKT J., Une description orientée objet des objets interactifs abstraits utilisés en Interface Homme-Machine, FUNDP, rapport interne,7 Octobre 1992.

[Somm92] SOMMERVILLE I., Le génie logiciel, Addison-Wesley, 1992.

[Varh94] VAHROL P., Visual Programming's Many Faces, Byte, Juillet 1994.

[Voss91] VOSS G., Object-Oriented Programming : An Introduction, Mc Graw-Hill, 1991.

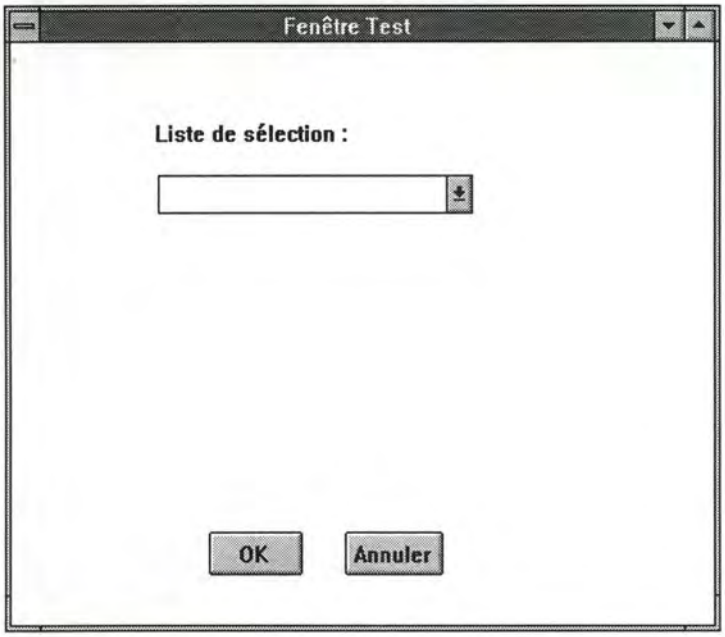
[Wils88] WILSON J., ROSENBERG D., Rapid Prototyping for User Interface Design, Chapitre 39 dans Handbook of Human-Computer Interaction, edité par M. Helander, North-Holland, 1988.

[Wirf90] WIRFS-BROCK R., JOHNSON R., Surveying current research in Object-Oriented design, Communications of the ACM, Vol 33, #9, Septembre 1990.

Annexes

Annexes

Annexe 1 : une fenêtre exportée sous forme de fichier texte




```
$PBExportHeader$w_test.srw
forward
global type w_test from Window
end type
type st_1 from statictext within w_test
end type
type ddlb_1 from dropdownlistbox within w_test
end type
type cb_2 from commandbutton within w_test
end type
type cb_1 from commandbutton within w_test
end type
end forward

global type w_test from Window
int X=750
```

```
int Y=401
int Width=2081
int Height=1545
boolean TitleBar=true
string Title="Fenêtre Test"
boolean ControlMenu=true
boolean MinBox=true
boolean MaxBox=true
boolean Resizable=true
st_1 st_1
ddlb_1 ddlb_1
cb_2 cb_2
cb_1 cb_1
end type
global w_test w_test

on w_test.create
this.st_1=create st_1
this.ddlb_1=create ddlb_1
this.cb_2=create cb_2
this.cb_1=create cb_1
this.Control[]={ this.st_1,&
this.ddlb_1,&
this.cb_2,&
this.cb_1}
end on

on w_test.destroy
destroy(st_1)
destroy(ddlb_1)
destroy(cb_2)
destroy(cb_1)
end on

type st_1 from statictext within w_test
int X=412
int Y=193
int Width=577
int Height=73
boolean Enabled=false
string Text="Liste de sélection :"
Alignment Alignment=Center!
```

```
long BackColor=16777215
end type

type ddlb_1 from dropdownlistbox within w_test
int X=426
int Y=329
int Width=919
int Height=509
int TabOrder=10
boolean VScrollBar=true
long BackColor=16777215
end type

type cb_2 from commandbutton within w_test
int X=970
int Y=1205
int Width=289
int Height=109
int TabOrder=30
string Text="Annuler"
end type

type cb_1 from commandbutton within w_test
int X=572
int Y=1205
int Width=275
int Height=109
int TabOrder=20
string Text="OK"
end type
```

Annexe 2 : le modèle d'une datawindow

A Le texte d'une datawindow exportée de PowerBuilder

```
$PBExportHeader$d_client.srd
release 3;
datawindow(units=0 timer_interval=0 color=1073741824 processing=0
print.documentname="" print.orientation = 0 print.margin.left = 110
```

```
print.margin.right = 110 print.margin.top = 97 print.margin.bottom = 97
print.paper.source = 0 print.paper.size = 0 print.prompt=no )
header(height=1 color="536870912")
summary(height=5 color="536870912")
footer(height=1 color="536870912")
detail(height=497 color="536870912")
table(column=(type=char(20) name=client_nom_cli dbname="client.nom_cli" )
 column=(type=char(20) name=client_prenom_cli dbname="client.prenom_cli"
)
 column=(type=char(13) name=client_tel_cli dbname="client.tel_cli" )
 column=(type=char(20) name=localite_localite dbname="localite.localite" )
 retrieve="PBSELECT(TABLE(NAME=~"client~" )
TABLE(NAME=~"localite~" ) COLUMN(NAME=~"client.nom_cli~")
COLUMN(NAME=~"client.prenom_cli~")
COLUMN(NAME=~"client.tel_cli~") COLUMN(NAME=~"localite.localite~")
JOIN (LEFT=~"localite.code_postal~" OP
=~"=~"RIGHT=~"client.code_postal_cli~" )) " )
text(band=detail color="0" alignment="1" border="0" x="37" y="4"
height="61" width="503" text="Nom Cli:" name=client_nom_cli_t
font.face="System" font.height="-9" font.weight="700" font.family="2"
font.pitch="1" font.charset="0" background.mode="1"
background.color="536870912")
column(band=detail id=1 border="1" alignment="0" color="0" height="61"
tabsequence=32766 width="947" x="567" y="4"  name=client_nom_cli
format="[general]" edit.limit=20 edit.case=any edit.focusrectangle=no
edit.autoselect=yes edit.autohscroll=yes  font.face="System" font.height="-9"
font.weight="700" font.family="2" font.pitch="1" font.charset="0"
background.mode="1" background.color="536870912")
text(band=detail color="0" alignment="1" border="0" x="37" y="128"
height="61" width="503" text="Prenom Cli:" name=client_prenom_cli_t
font.face="System" font.height="-9" font.weight="700" font.family="2"
font.pitch="1" font.charset="0" background.mode="1"
background.color="536870912")
column(band=detail id=2 border="1" alignment="0" color="0" height="61"
tabsequence=32766 width="947" x="567" y="128"  name=client_prenom_cli
format="[general]" edit.limit=20 edit.case=any edit.focusrectangle=no
edit.autoselect=yes edit.autohscroll=yes  font.face="System" font.height="-9"
font.weight="700" font.family="2" font.pitch="1" font.charset="0"
background.mode="1" background.color="536870912")
text(band=detail color="0" alignment="1" border="0" x="37" y="248"
height="61" width="503" text="Tel Cli:" name=client_tel_cli_t
font.face="System" font.height="-9" font.weight="700" font.family="2"
```

```
font.pitch="1" font.charset="0" background.mode="1"
background.color="536870912")
column(band=detail id=3 border="1" alignment="0" color="0" height="61"
tabsequence=32766 width="659" x="567" y="248"  name=client_tel_cli
format="[general]" edit.limit=13 edit.case=any edit.focusrectangle=no
edit.autoselect=yes edit.autohscroll=yes  font.face="System" font.height="-9"
font.weight="700" font.family="2" font.pitch="1" font.charset="0"
background.mode="1" background.color="536870912")
text(band=detail color="0" alignment="1" border="0" x="37" y="372"
height="61" width="503" text="Localite:" name=localite_localite_t
font.face="System" font.height="-9" font.weight="700" font.family="2"
font.pitch="1" font.charset="0" background.mode="1"
background.color="536870912")
column(band=detail id=4 border="1" alignment="0" color="0" height="61"
tabsequence=32766 width="947" x="567" y="372"  name=localite_localite
format="[general]" edit.limit=20 edit.case=any edit.focusrectangle=no
edit.autoselect=yes edit.autohscroll=yes  font.face="System" font.height="-9"
font.weight="700" font.family="2" font.pitch="1" font.charset="0"
background.mode="1" background.color="536870912")
```

B Le texte d'un modèle de datawindow

```
$PBExportHeader$d_<<CLEFON>>_<<CLESCH>>_free_f
$PBExportComments$Freeform of <<NOMSCH>>
release 3;
datawindow(units=0 timer_interval=0 color=12632256 processing=0
print.documentname="" print.orientation = 0 print.margin.left = 110
print.margin.right = 110 print.margin.top = 97 print.margin.bottom = 97
print.paper.source = 0 print.paper.size = 0 print.prompt=no )
header(height=800 color="536870912")
summary(height=1 color="536870912")
footer(height=1 color="536870912")
detail(height=1 color="536870912")
table(<<$[[
]]column=(type=<<FORCST>> update=yes name=<<NTXCST>>
dbname="<<NDBCST>>" )$>>
 column=(type=char(1) name=selected dbname="selected" )
 retrieve="SELECT <<$[[,]]<<NDBCST>>$>>,' ' FROM <<NDBTBL>>
<<WHESCH>>" )
<<$[[
```

]]text(band=header color="0" alignment="0" border="0" x="37"
y="<<YPOCST>>" height="65" width="<<WIDCST>>"
text="<<NOMCST>>" name=<<NTXCST>>_t font.face="Arial"
font.height="-10" font.weight="400" font.family="2" font.pitch="2"
font.charset="0" background.mode="1" background.color="536870912")
column(band=header id=<<ORDCST>> border="5" alignment="0" color="0"
height="65" tabsequence=<<ORDCST>> width="<<WIDCST>>" x="700"
y="<<YPOCST>>" name=<<NTXCST>> format="[general]" edit.limit=30
edit.case=any edit.focusrectangle=no edit.autoselect=yes edit.autohscroll=yes
font.face="Arial" font.height="-10" font.weight="400" font.family="2"
font.pitch="2" font.charset="0" background.mode="1"
background.color="536870912")$>>